

Table of Contents

Overview

[What is DocumentDB?](#)

[Core concepts](#)

[Global distribution](#)

[Scenarios](#)

[Common use cases](#)

[Going social with DocumentDB](#)

[Multi-tenancy](#)

Get Started

[Write your first app](#)

[.NET console app](#)

[.NET Core console app](#)

[Node.js console app](#)

[C++ console app](#)

[Build a web app](#)

[.NET web app](#)

[Node.js web app](#)

[Java web app](#)

[Python Flask web app](#)

[Develop Locally](#)

[FAQ](#)

How To

[Plan](#)

[Storage and performance](#)

[Partitioning and scaling](#)

[Consistency](#)

[NoSQL vs SQL](#)

[Manage](#)

[Import your data](#)

Model your data

Use geospatial data

Develop for multi-regions

Expire data automatically

Customize your indexes

Secure access to data

Back up and restore

Performance levels

Resource quotas

Increase quotas

Request units

Azure CLI and Azure Resource Manager

Firewall support

Supercharge your account

Develop

SQL query

Stored procedures, triggers, and UDFs

Performance testing

Performance tips

DocumentDB for MongoDB developers

Use the portal

Create a database account

Create a collection

Add global replication

Add and edit documents

Query documents

Manage an account

Monitor an account

Manage scripts

Troubleshooting tips

Integrate

Deploy a website with Azure App Service

[Application logging with Logic Apps](#)

[Bind to Azure Functions](#)

[Analyze data with Hadoop](#)

[Integrate with Azure Search](#)

[Move data with Azure Data Factory](#)

[Analyze real-time data with Azure Stream Analytics](#)

[Get notifications with Logic Apps](#)

[Process sensor data in real time](#)

[Visualize your data with Power BI](#)

Reference

[Java SDK](#)

[.NET SDK](#)

[.NET Core SDK](#)

[.NET samples](#)

[Node.js SDK](#)

[Node.js samples](#)

[Python SDK](#)

[Python samples](#)

[SQL](#)

[SQL grammar cheat sheet](#)

[REST](#)

[REST Resource Provider](#)

Resources

[Pricing](#)

[MSDN forum](#)

[Stack Overflow](#)

[Videos](#)

[Service updates](#)

[Community portal](#)

[Query Playground](#)

[Schema agnostic indexing paper](#)

[Data consistency explained through baseball](#)

Book: Using Microsoft Azure DocumentDB in a Node.js Application

Learning path

Introduction to DocumentDB: A NoSQL JSON Database

11/22/2016 • 7 min to read • [Edit on GitHub](#)

Contributors

[mimig](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [arramac](#) • [Andrew Liu](#) • [dabutvin](#) • [v-aljenk](#)

What is DocumentDB?

DocumentDB is a fully managed NoSQL database service built for fast and predictable performance, high availability, elastic scaling, global distribution, and ease of development. As a schema-free NoSQL database, DocumentDB provides rich and familiar SQL query capabilities with consistent low latencies on JSON data - ensuring that 99% of your reads are served under 10 milliseconds and 99% of your writes are served under 15 milliseconds. These unique benefits make DocumentDB a great fit for web, mobile, gaming, and IoT, and many other applications that need seamless scale and global replication.

How can I learn about DocumentDB?

A quick way to learn about DocumentDB and see it in action is to follow these three steps:

1. Watch the two minute [What is DocumentDB?](#) video, which introduces the benefits of using DocumentDB.
2. Watch the three minute [Create DocumentDB on Azure](#) video, which highlights how to get started with DocumentDB by using the Azure Portal.
3. Visit the [Query Playground](#), where you can walk through different activities to learn about the rich querying functionality available in DocumentDB. Then, head over to the Sandbox tab and run your own custom SQL queries and experiment with DocumentDB.

Then, return to this article, where we'll dig in deeper.

What capabilities and key features does DocumentDB offer?

Azure DocumentDB offers the following key capabilities and benefits:

- **Elastically scalable throughput and storage:** Easily scale up or scale down your DocumentDB JSON database to meet your application needs. Your data is stored on solid state disks (SSD) for low predictable latencies. DocumentDB supports containers for storing JSON data called collections that can scale to virtually unlimited storage sizes and provisioned throughput. You can elastically scale DocumentDB with predictable performance seamlessly as your application grows.
- **Multi-region replication:** DocumentDB transparently replicates your data to all regions you've associated with your DocumentDB account, enabling you to develop applications that require global access to data while providing tradeoffs between consistency, availability and performance, all with corresponding guarantees. DocumentDB provides transparent regional failover with multi-homing APIs, and the ability to elastically scale throughput and storage across the globe. Learn more in [Distribute data globally with DocumentDB](#).
- **Ad hoc queries with familiar SQL syntax:** Store heterogeneous JSON documents within DocumentDB and query these documents through a familiar SQL syntax. DocumentDB utilizes a highly concurrent, lock free, log structured indexing technology to automatically index all document content. This enables rich real-time queries without the need to specify schema hints, secondary indexes, or views. Learn more in [Query DocumentDB](#).
- **JavaScript execution within the database:** Express application logic as stored procedures, triggers, and

user defined functions (UDFs) using standard JavaScript. This allows your application logic to operate over data without worrying about the mismatch between the application and the database schema. DocumentDB provides full transactional execution of JavaScript application logic directly inside the database engine. The deep integration of JavaScript enables the execution of INSERT, REPLACE, DELETE, and SELECT operations from within a JavaScript program as an isolated transaction. Learn more in [DocumentDB server-side programming](#).

- **Tunable consistency levels:** Select from four well defined consistency levels to achieve optimal trade-off between consistency and performance. For queries and read operations, DocumentDB offers four distinct consistency levels: strong, bounded-staleness, session, and eventual. These granular, well-defined consistency levels allow you to make sound trade-offs between consistency, availability, and latency. Learn more in [Using consistency levels to maximize availability and performance in DocumentDB](#).
- **Fully managed:** Eliminate the need to manage database and machine resources. As a fully-managed Microsoft Azure service, you do not need to manage virtual machines, deploy and configure software, manage scaling, or deal with complex data-tier upgrades. Every database is automatically backed up and protected against regional failures. You can easily add a DocumentDB account and provision capacity as you need it, allowing you to focus on your application instead of operating and managing your database.
- **Open by design:** Get started quickly by using existing skills and tools. Programming against DocumentDB is simple, approachable, and does not require you to adopt new tools or adhere to custom extensions to JSON or JavaScript. You can access all of the database functionality including CRUD, query, and JavaScript processing over a simple RESTful HTTP interface. DocumentDB embraces existing formats, languages, and standards while offering high value database capabilities on top of them.
- **Automatic indexing:** By default, DocumentDB [automatically indexes](#) all the documents in the database and does not expect or require any schema or creation of secondary indices. Don't want to index everything? Don't worry, you can [opt out of paths in your JSON files](#) too.

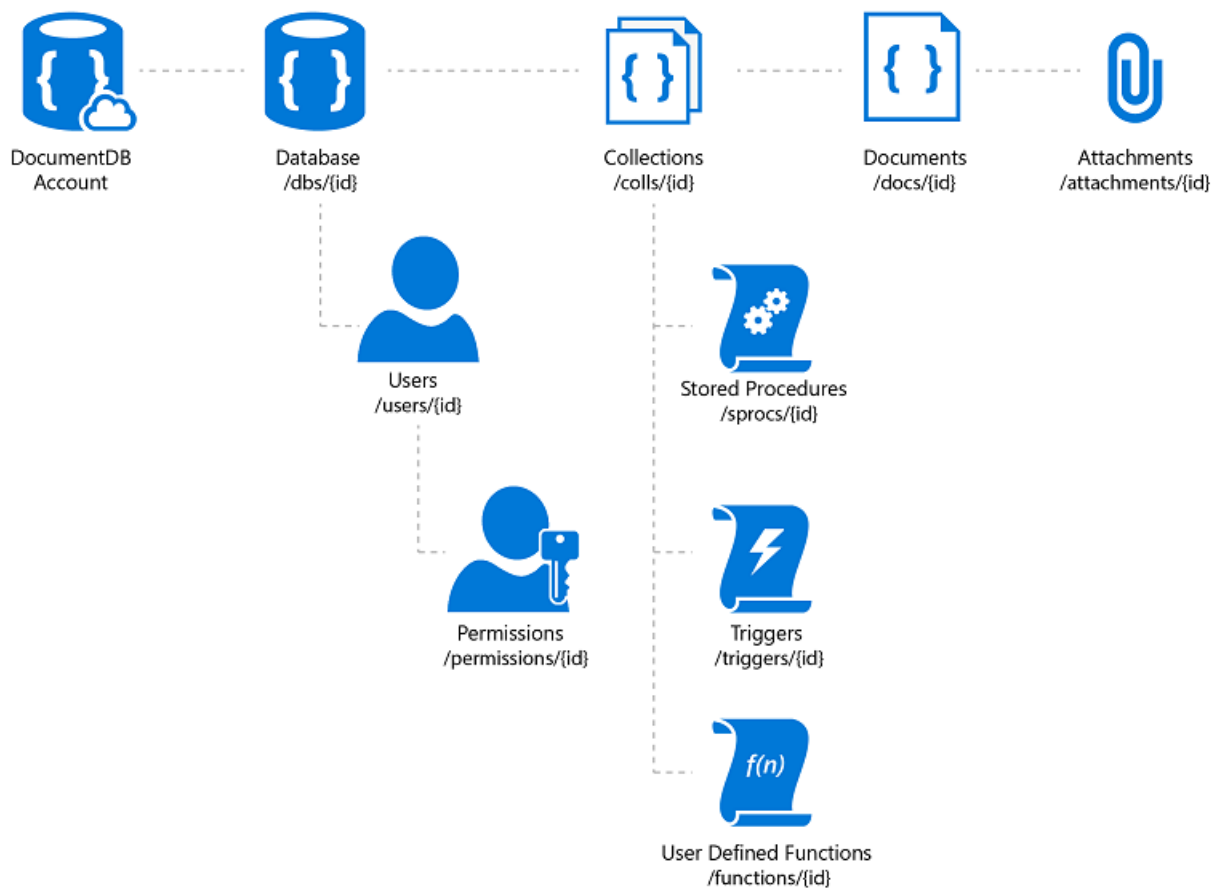
How does DocumentDB manage data?

Azure DocumentDB manages JSON data through well-defined database resources. These resources are replicated for high availability and are uniquely addressable by their logical URI. DocumentDB offers a simple HTTP based RESTful programming model for all resources.

The DocumentDB database account is a unique namespace that gives you access to Azure DocumentDB. Before you can create a database account, you must have an Azure subscription, which gives you access to a variety of Azure services.

All resources within DocumentDB are modeled and stored as JSON documents. Resources are managed as items, which are JSON documents containing metadata, and as feeds which are collections of items. Sets of items are contained within their respective feeds.

The image below shows the relationships between the DocumentDB resources:



A database account consists of a set of databases, each containing multiple collections, each of which can contain stored procedures, triggers, UDFs, documents, and related attachments. A database also has associated users, each with a set of permissions to access various other collections, stored procedures, triggers, UDFs, documents, or attachments. While databases, users, permissions, and collections are system-defined resources with well-known schemas - documents, stored procedures, triggers, UDFs, and attachments contain arbitrary, user defined JSON content.

How can I develop apps with DocumentDB?

Azure DocumentDB exposes resources through a REST API that can be called by any language capable of making HTTP/HTTPS requests. Additionally, DocumentDB offers programming libraries for several popular languages. These libraries simplify many aspects of working with Azure DocumentDB by handling details such as address caching, exception management, automatic retries and so forth. Libraries are currently available for the following languages and platforms:

DOWNLOAD	DOCUMENTATION
.NET SDK	.NET library
Node.js SDK	Node.js library
Java SDK	Java library
JavaScript SDK	JavaScript library
n/a	Server-side JavaScript SDK
Python SDK	Python library

Using the [Azure DocumentDB Emulator](#), you can develop and test your application locally, without creating an Azure subscription or incurring any costs. When you're satisfied with how your application is working in the DocumentDB Emulator, you can switch to using an Azure DocumentDB account in the cloud.

Beyond basic create, read, update, and delete operations, DocumentDB provides a rich SQL query interface for retrieving JSON documents and server side support for transactional execution of JavaScript application logic. The query and script execution interfaces are available through all platform libraries as well as the REST APIs.

SQL query

Azure DocumentDB supports querying documents using a SQL language, which is rooted in the JavaScript type system, and expressions with support for relational, hierarchical, and spatial queries. The DocumentDB query language is a simple yet powerful interface to query JSON documents. The language supports a subset of ANSI SQL grammar and adds deep integration of JavaScript object, arrays, object construction, and function invocation. DocumentDB provides its query model without any explicit schema or indexing hints from the developer.

User Defined Functions (UDFs) can be registered with DocumentDB and referenced as part of a SQL query, thereby extending the grammar to support custom application logic. These UDFs are written as JavaScript programs and executed within the database.

For .NET developers, DocumentDB also offers a LINQ query provider as part of the [.NET SDK](#).

Transactions and JavaScript execution

DocumentDB allows you to write application logic as named programs written entirely in JavaScript. These programs are registered for a collection and can issue database operations on the documents within a given collection. JavaScript can be registered for execution as a trigger, stored procedure or user defined function. Triggers and stored procedures can create, read, update, and delete documents whereas user defined functions execute as part of the query execution logic without write access to the collection.

JavaScript execution within DocumentDB is modeled after the concepts supported by relational database systems, with JavaScript as a modern replacement for Transact-SQL. All JavaScript logic is executed within an ambient ACID transaction with snapshot isolation. During the course of its execution, if the JavaScript throws an exception, then the entire transaction is aborted.

Next steps

Already have an Azure account? Then you can get started with DocumentDB in the [Azure Portal](#) by [creating a DocumentDB database account](#).

Don't have an Azure account? You can:

- Sign up for an [Azure free trial](#), which gives you 30 days and \$200 to try all the Azure services.
- If you have an MSDN subscription, you are eligible for [\\$150 in free Azure credits per month](#) to use on any Azure service.
- Download the [Azure DocumentDB Emulator](#) to develop your application locally.

Then, when you're ready to learn more, visit our [learning path](#) to navigate all the learning resources available to you.

DocumentDB hierarchical resource model and concepts

11/15/2016 • 24 min to read • [Edit on GitHub](#)

Contributors

[Andrew Hoh](#) • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [mimig](#) • [arramac](#) • [v-aljenk](#) • [Dene Hager](#)

The database entities that DocumentDB manages are referred to as **resources**. Each resource is uniquely identified by a logical URI. You can interact with the resources using standard HTTP verbs, request/response headers and status codes.

By reading this article, you'll be able to answer the following questions:

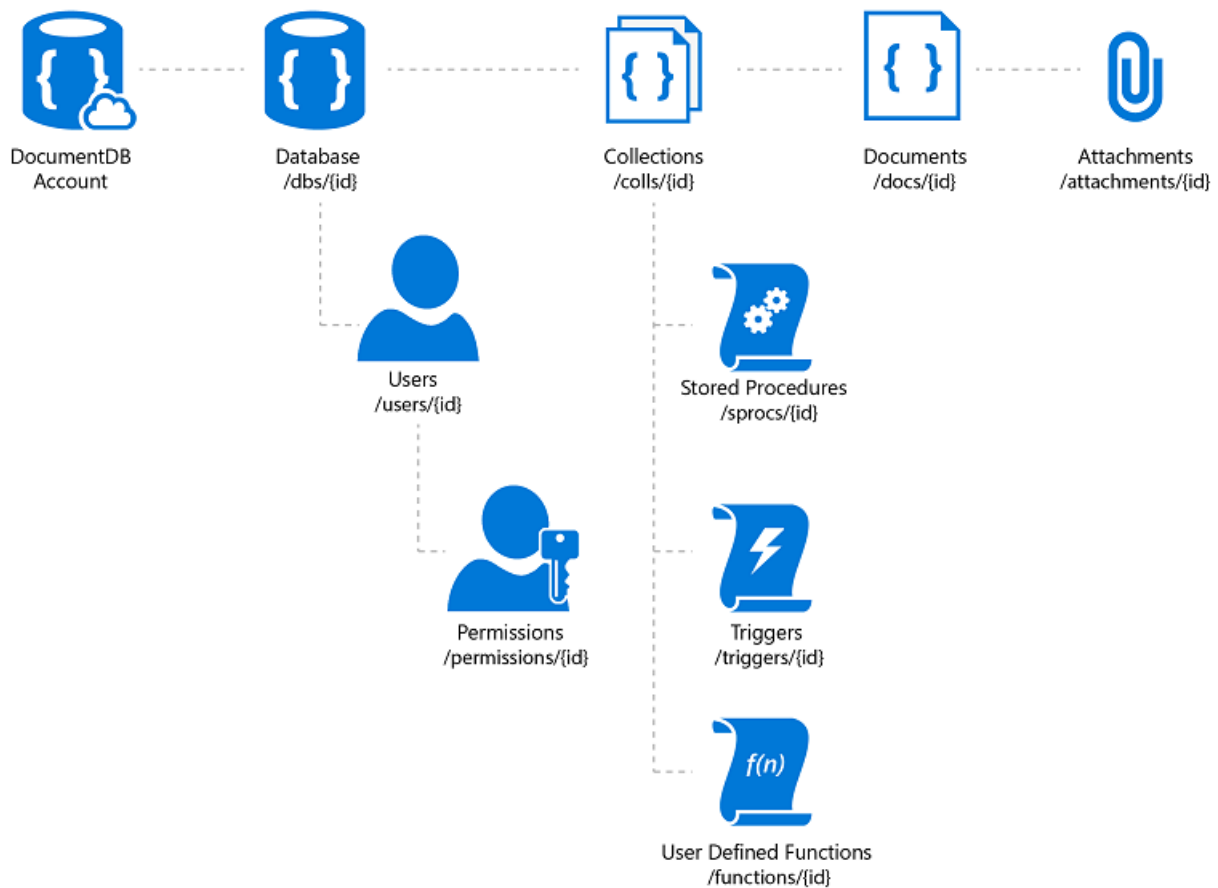
- What is DocumentDB's resource model?
- What are system defined resources as opposed to user defined resources?
- How do I address a resource?
- How do I work with collections?
- How do I work with stored procedures, triggers and User Defined Functions (UDFs)?

Hierarchical resource model

As the following diagram illustrates, the DocumentDB hierarchical **resource model** consists of sets of resources under a database account, each addressable via a logical and stable URI. A set of resources will be referred to as a **feed** in this article.

NOTE

DocumentDB offers a highly efficient TCP protocol which is also RESTful in its communication model, available through the [.NET client SDK](#).



Hierarchical resource model

To start working with resources, you must [create a DocumentDB database account](#) using your Azure subscription. A database account can consist of a set of **databases**, each containing multiple **collections**, each of which in turn contain **stored procedures**, **triggers**, **UDFs**, **documents** and related **attachments** (preview feature). A database also has associated **users**, each with a set of **permissions** to access collections, stored procedures, triggers, UDFs, documents or attachments. While databases, users, permissions and collections are system-defined resources with well-known schemas, documents and attachments contain arbitrary, user defined JSON content.

RESOURCE	DESCRIPTION
Database account	A database account is associated with a set of databases and a fixed amount of blob storage for attachments (preview feature). You can create one or more database accounts using your Azure subscription. For more information, visit our pricing page .
Database	A database is a logical container of document storage partitioned across collections. It is also a users container.
User	The logical namespace for scoping permissions.
Permission	An authorization token associated with a user for access to a specific resource.
Collection	A collection is a container of JSON documents and the associated JavaScript application logic. A collection is a billable entity, where the cost is determined by the performance level associated with the collection. Collections can span one or more partitions/servers and can scale to handle practically unlimited volumes of storage or throughput.

RESOURCE	DESCRIPTION
Stored Procedure	Application logic written in JavaScript which is registered with a collection and transactionally executed within the database engine.
Trigger	Application logic written in JavaScript executed before or after either an insert, replace or delete operation.
UDF	Application logic written in JavaScript. UDFs enable you to model a custom query operator and thereby extend the core DocumentDB query language.
Document	User defined (arbitrary) JSON content. By default, no schema needs to be defined nor do secondary indices need to be provided for all the documents added to a collection.
(Preview) Attachment	An attachment is a special document containing references and associated metadata for external blob/media. The developer can choose to have the blob managed by DocumentDB or store it with an external blob service provider such as OneDrive, Dropbox, etc.

System vs. user defined resources

Resources such as database accounts, databases, collections, users, permissions, stored procedures, triggers, and UDFs - all have a fixed schema and are called system resources. In contrast, resources such as documents and attachments have no restrictions on the schema and are examples of user defined resources. In DocumentDB, both system and user defined resources are represented and managed as standard-compliant JSON. All resources, system or user defined, have the following common properties.

NOTE

Note that all system generated properties in a resource are prefixed with an underscore () in their JSON representation.

Property	User settable or system generated?	Purpose
_rid	System generated	System generated, unique and hierarchical identifier of the resource
_etag	System generated	etag of the resource required for optimistic concurrency control
_ts	System generated	Last updated timestamp of the resource
_self	System generated	Unique addressable URI of the resource

id	System generated	User defined unique name of the resource (with the same partition key value). If the user does not specify an id, an id will be system generated
----	------------------	--

Wire representation of resources

DocumentDB does not mandate any proprietary extensions to the JSON standard or special encodings; it works with standard compliant JSON documents.

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 <code>_SELF</code>	DESCRIPTION
<code>/dbs</code>	Feed of databases under a database account
<code>/dbs/{dbName}</code>	Database with an id matching the value <code>{dbName}</code>
<code>/dbs/{dbName}/colls/</code>	Feed of collections under a database
<code>/dbs/{dbName}/colls/{collName}</code>	Collection with an id matching the value <code>{collName}</code>
<code>/dbs/{dbName}/colls/{collName}/docs</code>	Feed of documents under a collection
<code>/dbs/{dbName}/colls/{collName}/docs/{docId}</code>	Document with an id matching the value <code>{doc}</code>
<code>/dbs/{dbName}/users/</code>	Feed of users under a database
<code>/dbs/{dbName}/users/{userId}</code>	User with an id matching the value <code>{user}</code>
<code>/dbs/{dbName}/users/{userId}/permissions</code>	Feed of permissions under a user
<code>/dbs/{dbName}/users/{userId}/permissions/{permissionId}</code>	Permission with an id matching the value <code>{permission}</code>

Each resource has a unique user defined name exposed via the `id` property. Note: for documents, if the user does not specify an id, our supported SDKs will automatically generate a unique id for the document. The `id` is a user defined string, of up to 256 characters that is unique within the context of a specific parent resource.

Each resource also has a system generated hierarchical resource identifier (also referred to as an RID), which is available via the `_rid` property. The RID encodes the entire hierarchy of a given resource and it is a convenient internal representation used to enforce referential integrity in a distributed manner. The RID is unique within a database account and it is internally used by DocumentDB for efficient routing without requiring cross partition lookups. The values of the `_self` and the `_rid` properties are both alternate and canonical representations of a resource.

The DocumentDB REST APIs support addressing of resources and routing of requests by both the `id` and the `_rid` properties.

Database accounts

You can provision one or more DocumentDB database accounts using your Azure subscription.

You can [create and manage DocumentDB database accounts](http://portal.azure.com/) via the Azure Portal at <http://portal.azure.com/>. Creating and managing a database account requires administrative access and can only be performed under your Azure subscription.

Database account properties

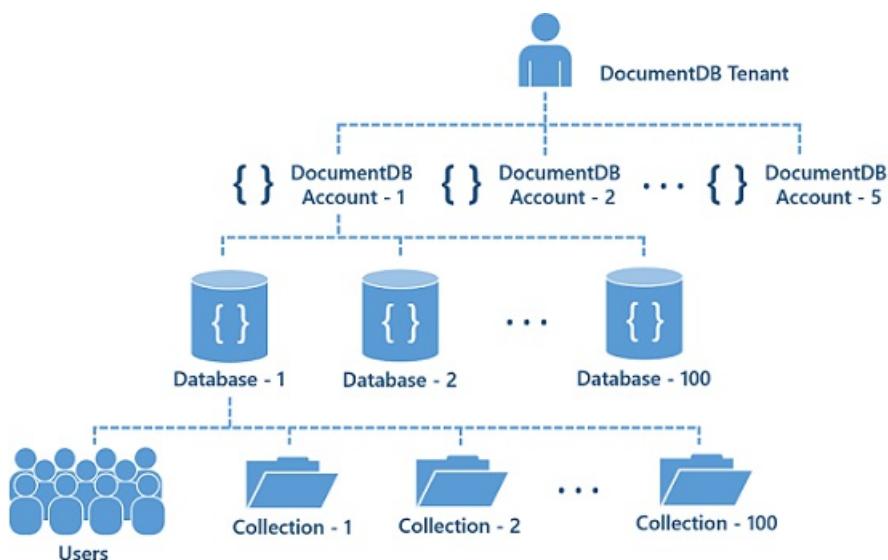
As part of provisioning and managing a database account you can configure and read the following properties:

Property Name	Description
Consistency Policy	<p>Set this property to configure the default consistency level for all the collections under your database account. You can override the consistency level on a per request basis using the [x-ms-consistency-level] request header.</p> <p>Note that this property only applies to the <i>user defined resources</i>. All system defined resources are configured to support reads/queries with strong consistency.</p>
Authorization Keys	<p>These are the primary and secondary master and readonly keys that provide administrative access to all of the resources under the database account.</p>

Note that in addition to provisioning, configuring and managing your database account from the Azure Portal, you can also programmatically create and manage DocumentDB database accounts by using the [Azure DocumentDB REST APIs](#) as well as [client SDKs](#).

Databases

A DocumentDB database is a logical container of one or more collections and users, as shown in the following diagram. You can create any number of databases under a DocumentDB database account subject to offer limits.



A Database is a logical container of users and collections

A database can contain virtually unlimited document storage partitioned by collections, which form the transaction domains for the documents contained within them.

Elastic scale of a DocumentDB database

A DocumentDB database is elastic by default – ranging from a few GB to petabytes of SSD backed document storage and provisioned throughput.

Unlike a database in traditional RDBMS, a database in DocumentDB is not scoped to a single machine. With DocumentDB, as your application's scale needs to grow, you can create more collections, databases, or both. Indeed, various first party applications within Microsoft have been using DocumentDB at a consumer scale by creating extremely large DocumentDB databases each containing thousands of collections with terabytes of document storage. You can grow or shrink a database by adding or removing collections to meet your application's scale requirements.

You can create any number of collections within a database subject to the offer. Each collection has SSD backed storage and throughput provisioned for you depending on the selected performance tier.

A DocumentDB database is also a container of users. A user, in-turn, is a logical namespace for a set of permissions that provides fine-grained authorization and access to collections, documents and attachments.

As with other resources in the DocumentDB resource model, databases can be created, replaced, deleted, read or enumerated easily using either [Azure DocumentDB REST APIs](#) or any of the [client SDKs](#). DocumentDB guarantees strong consistency for reading or querying the metadata of a database resource. Deleting a database automatically ensures that you cannot access any of the collections or users contained within it.

Collections

A DocumentDB collection is a container for your JSON documents. A collection is also a unit of scale for transactions and queries.

Elastic SSD backed document storage

A collection is intrinsically elastic - it automatically grows and shrinks as you add or remove documents. Collections are logical resources and can span one or more physical partitions or servers. The number of partitions within a collection is determined by DocumentDB based on the storage size and the provisioned throughput of your collection. Every partition in DocumentDB has a fixed amount of SSD-backed storage associated with it, and is replicated for high availability. Partition management is fully managed by Azure DocumentDB, and you do not have to write complex code or manage your partitions. DocumentDB collections are **practically unlimited** in terms of storage and throughput.

Automatic indexing of collections

DocumentDB is a true schema-free database system. It does not assume or require any schema for the JSON documents. As you add documents to a collection, DocumentDB automatically indexes them and they are available for you to query. Automatic indexing of documents without requiring schema or secondary indexes is a key capability of DocumentDB and is enabled by write-optimized, lock-free and log-structured index maintenance techniques. DocumentDB supports sustained volume of extremely fast writes while still serving consistent queries. Both document and index storage are used to calculate the storage consumed by each collection. You can control the storage and performance trade-offs associated with indexing by configuring the indexing policy for a collection.

Configuring the indexing policy of a collection

The indexing policy of each collection allows you to make performance and storage trade-offs associated with indexing. The following options are available to you as part of indexing configuration:

- Choose whether the collection automatically indexes all of the documents or not. By default, all documents are automatically indexed. You can choose to turn off automatic indexing and selectively add only specific documents to the index. Conversely, you can selectively choose to exclude only specific documents. You can achieve this by setting the automatic property to be true or false on the indexingPolicy of a collection and using the [x-ms-indexingdirective] request header while inserting, replacing or deleting a document.
- Choose whether to include or exclude specific paths or patterns in your documents from the index. You can achieve this by setting includedPaths and excludedPaths on the indexingPolicy of a collection respectively. You can also configure the storage and performance trade-offs for range and hash queries for specific path patterns.

- Choose between synchronous (consistent) and asynchronous (lazy) index updates. By default, the index is updated synchronously on each insert, replace or delete of a document to the collection. This enables the queries to honor the same consistency level as that of the document reads. While DocumentDB is write optimized and supports sustained volumes of document writes along with synchronous index maintenance and serving consistent queries, you can configure certain collections to update their index lazily. Lazy indexing boosts the write performance further and is ideal for bulk ingestion scenarios for primarily read-heavy collections.

The indexing policy can be changed by executing a PUT on the collection. This can be achieved either through the [client SDK](#), the [Azure Portal](#) or the [Azure DocumentDB REST APIs](#).

Querying a collection

The documents within a collection can have arbitrary schemas and you can query documents within a collection without providing any schema or secondary indices upfront. You can query the collection using the [DocumentDB SQL syntax](#), which provides rich hierarchical, relational, and spatial operators and extensibility via JavaScript-based UDFs. JSON grammar allows for modeling JSON documents as trees with labels as the tree nodes. This is exploited both by DocumentDB's automatic indexing techniques as well as DocumentDB's SQL dialect. The DocumentDB query language consists of three main aspects:

1. A small set of query operations that map naturally to the tree structure including hierarchical queries and projections.
2. A subset of relational operations including composition, filter, projections, aggregates and self joins.
3. Pure JavaScript based UDFs that work with (1) and (2).

The DocumentDB query model attempts to strike a balance between functionality, efficiency and simplicity. The DocumentDB database engine natively compiles and executes the SQL query statements. You can query a collection using the [Azure DocumentDB REST APIs](#) or any of the [client SDKs](#). The .NET SDK comes with a LINQ provider.

TIP

You can try out DocumentDB and run SQL queries against our dataset in the [Query Playground](#).

Multi-document transactions

Database transactions provide a safe and predictable programming model for dealing with concurrent changes to the data. In RDBMS, the traditional way to write business logic is to write **stored-procedures** and/or **triggers** and ship it to the database server for transactional execution. In RDBMS, the application programmer is required to deal with two disparate programming languages:

- The (non-transactional) application programming language (e.g. JavaScript, Python, C#, Java, etc.)
- T-SQL, the transactional programming language which is natively executed by the database

By virtue of its deep commitment to JavaScript and JSON directly within the database engine, DocumentDB provides an intuitive programming model for executing JavaScript based application logic directly on the collections in terms of stored procedures and triggers. This allows for both of the following:

- Efficient implementation of concurrency control, recovery, automatic indexing of the JSON object graphs directly in the database engine
- Naturally expressing control flow, variable scoping, assignment and integration of exception handling primitives with database transactions directly in terms of the JavaScript programming language

The JavaScript logic registered at a collection level can then issue database operations on the documents of the given collection. DocumentDB implicitly wraps the JavaScript based stored procedures and triggers within an ambient ACID transactions with snapshot isolation across documents within a collection. During the course of its

execution, if the JavaScript throws an exception, then the entire transaction is aborted. The resulting programming model is a very simple yet powerful. JavaScript developers get a “durable” programming model while still using their familiar language constructs and library primitives.

The ability to execute JavaScript directly within the database engine in the same address space as the buffer pool enables performant and transactional execution of database operations against the documents of a collection. Furthermore, DocumentDB database engine makes a deep commitment to the JSON and JavaScript eliminates any impedance mismatch between the type systems of application and the database.

After creating a collection, you can register stored procedures, triggers and UDFs with a collection using the [Azure DocumentDB REST APIs](#) or any of the [client SDKs](#). After registration, you can reference and execute them. Consider the following stored procedure written entirely in JavaScript, the code below takes two arguments (book name and author name) and creates a new document, queries for a document and then updates it – all within an implicit ACID transaction. At any point during the execution, if a JavaScript exception is thrown, the entire transaction aborts.

```
function businessLogic(name, author) {
    var context = getContext();
    var collectionManager = context.getCollection();
    var collectionLink = collectionManager.getSelfLink()

    // create a new document.
    collectionManager.createDocument(collectionLink,
        {id: name, author: author},
        function(err, documentCreated) {
            if(err) throw new Error(err.message);

            // filter documents by author
            var filterQuery = "SELECT * from root r WHERE r.author = 'George R.'";
            collectionManager.queryDocuments(collectionLink,
                filterQuery,
                function(err, matchingDocuments) {
                    if(err) throw new Error(err.message);

                    context.getResponse().setBody(matchingDocuments.length);

                    // Replace the author name for all documents that satisfied the query.
                    for (var i = 0; i < matchingDocuments.length; i++) {
                        matchingDocuments[i].author = "George R. R. Martin";
                        // we don't need to execute a callback because they are in parallel
                        collectionManager.replaceDocument(matchingDocuments[i]._self,
                            matchingDocuments[i]);
                    }
                })
        })
};
```

The client can “ship” the above JavaScript logic to the database for transactional execution via HTTP POST. For more information about using HTTP methods, see [RESTful interactions with DocumentDB resources](#).

```
client.createStoredProcedureAsync(collection._self, {id: "CRUDProc", body: businessLogic})
    .then(function(createdStoredProcedure) {
        return client.executeStoredProcedureAsync(createdStoredProcedure.resource._self,
            "NoSQL Distilled",
            "Martin Fowler");
    })
    .then(function(result) {
        console.log(result);
    },
    function(error) {
        console.log(error);
    });
```

Notice that because the database natively understands JSON and JavaScript, there is no type system mismatch, no “OR mapping” or code generation magic required.

Stored procedures and triggers interact with a collection and the documents in a collection through a well-defined object model, which exposes the current collection context.

Collections in DocumentDB can be created, deleted, read or enumerated easily using either the [Azure DocumentDB REST APIs](#) or any of the [client SDKs](#). DocumentDB always provides strong consistency for reading or querying the metadata of a collection. Deleting a collection automatically ensures that you cannot access any of the documents, attachments, stored procedures, triggers, and UDFs contained within it.

Stored procedures, triggers and User Defined Functions (UDF)

As described in the previous section, you can write application logic to run directly within a transaction inside of the database engine. The application logic can be written entirely in JavaScript and can be modeled as a stored procedure, trigger or a UDF. The JavaScript code within a stored procedure or a trigger can insert, replace, delete, read or query documents within a collection. On the other hand, the JavaScript within a UDF cannot insert, replace, or delete documents. UDFs enumerate the documents of a query's result set and produce another result set. For multi-tenancy, DocumentDB enforces a strict reservation based resource governance. Each stored procedure, trigger or a UDF gets a fixed quantum of operating system resources to do its work. Furthermore, stored procedures, triggers or UDFs cannot link against external JavaScript libraries and are blacklisted if they exceed the resource budgets allocated to them. You can register, unregister stored procedures, triggers or UDFs with a collection by using the REST APIs. Upon registration a stored procedure, trigger, or a UDF is pre-compiled and stored as byte code which gets executed later. The following section illustrate how you can use the DocumentDB JavaScript SDK to register, execute, and unregister a stored procedure, trigger, and a UDF. The JavaScript SDK is a simple wrapper over the [DocumentDB REST APIs](#).

Registering a stored procedure

Registration of a stored procedure creates a new stored procedure resource on a collection via HTTP POST.

```
var storedProc = {
  id: "validateAndCreate",
  body: function (documentToCreate) {
    documentToCreate.id = documentToCreate.id.toUpperCase();

    var collectionManager = getContext().getCollection();
    collectionManager.createDocument(collectionManager.getSelfLink(),
      documentToCreate,
      function(err, documentCreated) {
        if(err) throw new Error('Error while creating document: ' + err.message);
        getContext().getResponse().setBody('success - created ' +
          documentCreated.name);
      });
  }
};

client.createStoredProcedureAsync(collection._self, storedProc)
  .then(function (createdStoredProcedure) {
    console.log("Successfully created stored procedure");
  }, function(error) {
    console.log("Error");
  });
```

Executing a stored procedure

Execution of a stored procedure is done by issuing an HTTP POST against an existing stored procedure resource by passing parameters to the procedure in the request body.

```

var inputDocument = {id : "document1", author: "G. G. Marquez"};
client.executeStoredProcedureAsync(createdStoredProcedure.resource._self, inputDocument)
    .then(function(executionResult) {
        assert.equal(executionResult, "success - created DOCUMENT1");
    }, function(error) {
        console.log("Error");
    });

```

Unregistering a stored procedure

Unregistering a stored procedure is simply done by issuing an HTTP DELETE against an existing stored procedure resource.

```

client.deleteStoredProcedureAsync(createdStoredProcedure.resource._self)
    .then(function (response) {
        return;
    }, function(error) {
        console.log("Error");
    });

```

Registering a pre-trigger

Registration of a trigger is done by creating a new trigger resource on a collection via HTTP POST. You can specify if the trigger is a pre or a post trigger and the type of operation it can be associated with (e.g. Create, Replace, Delete, or All).

```

var preTrigger = {
    id: "upperCaseId",
    body: function() {
        var item = getContext().getRequest().getBody();
        item.id = item.id.toUpperCase();
        getContext().getRequest().setBody(item);
    },
    triggerType: TriggerType.Pre,
    triggerOperation: TriggerOperation.All
}

client.createTriggerAsync(collection._self, preTrigger)
    .then(function (createdPreTrigger) {
        console.log("Successfully created trigger");
    }, function(error) {
        console.log("Error");
    });

```

Executing a pre-trigger

Execution of a trigger is done by specifying the name of an existing trigger at the time of issuing the POST/PUT/DELETE request of a document resource via the request header.

```

client.createDocumentAsync(collection._self, { id: "doc1", key: "Love in the Time of Cholera" }, {
    preTriggerInclude: "upperCaseId" })
    .then(function(createdDocument) {
        assert.equal(createdDocument.resource.id, "DOC1");
    }, function(error) {
        console.log("Error");
    });

```

Unregistering a pre-trigger

Unregistering a trigger is simply done via issuing an HTTP DELETE against an existing trigger resource.

```

client.deleteTriggerAsync(createdPreTrigger._self);
    .then(function(response) {
        return;
    }, function(error) {
        console.log("Error");
    });

```

Registering a UDF

Registration of a UDF is done by creating a new UDF resource on a collection via HTTP POST.

```

var udf = {
    id: "mathSqrt",
    body: function(number) {
        return Math.sqrt(number);
    },
};
client.createUserDefinedFunctionAsync(collection._self, udf)
    .then(function (createdUdf) {
        console.log("Successfully created stored procedure");
    }, function(error) {
        console.log("Error");
    });

```

Executing a UDF as part of the query

A UDF can be specified as part of the SQL query and is used as a way to extend the core [SQL query language of DocumentDB](#).

```

var filterQuery = "SELECT udf.mathSqrt(r.Age) AS sqrtAge FROM root r WHERE r.FirstName='John'";
client.queryDocuments(collection._self, filterQuery).toArrayAsync();
    .then(function(queryResponse) {
        var queryResponseDocuments = queryResponse.feed;
    }, function(error) {
        console.log("Error");
    });

```

Unregistering a UDF

Unregistering a UDF is simply done by issuing an HTTP DELETE against an existing UDF resource.

```

client.deleteUserDefinedFunctionAsync(createdUdf._self)
    .then(function(response) {
        return;
    }, function(error) {
        console.log("Error");
    });

```

Although the snippets above showed the registration (POST), unregistration (PUT), read/list (GET) and execution (POST) via the [DocumentDB JavaScript SDK](#), you can also use the [REST APIs](#) or other [client SDKs](#).

Documents

You can insert, replace, delete, read, enumerate and query arbitrary JSON documents in a collection. DocumentDB does not mandate any schema and does not require secondary indexes in order to support querying over documents in a collection.

Being a truly open database service, DocumentDB does not invent any specialized data types (e.g. date time) or specific encodings for JSON documents. Note that DocumentDB does not require any special JSON conventions to codify the relationships among various documents; the SQL syntax of DocumentDB provides very powerful hierarchical and relational query operators to query and project documents without any special annotations or

need to codify relationships among documents using distinguished properties.

As with all other resources, documents can be created, replaced, deleted, read, enumerated and queried easily using either REST APIs or any of the [client SDKs](#). Deleting a document instantly frees up the quota corresponding to all of the nested attachments. The read consistency level of documents follows the consistency policy on the database account. This policy can be overridden on a per-request basis depending on data consistency requirements of your application. When querying documents, the read consistency follows the indexing mode set on the collection. For “consistent”, this follows the account’s consistency policy.

Attachments and media

NOTE

Attachment and media resources are preview features.

DocumentDB allows you to store binary blobs/media either with DocumentDB or to your own remote media store. It also allows you to represent the metadata of a media in terms of a special document called attachment. An attachment in DocumentDB is a special (JSON) document that references the media/blob stored elsewhere. An attachment is simply a special document that captures the metadata (e.g. location, author etc.) of a media stored in a remote media storage.

Consider a social reading application which uses DocumentDB to store ink annotations, and metadata including comments, highlights, bookmarks, ratings, likes/dislikes etc. associated for an e-book of a given user.

- The content of the book itself is stored in the media storage either available as part of DocumentDB database account or a remote media store.
- An application may store each user’s metadata as a distinct document -- e.g. Joe’s metadata for book1 is stored in a document referenced by `/colls/joe/docs/book1`.
- Attachments pointing to the content pages of a given book of a user are stored under the corresponding document e.g. `/colls/joe/docs/book1/chapter1`, `/colls/joe/docs/book1/chapter2` etc.

Note that the examples listed above use friendly ids to convey the resource hierarchy. Resources are accessed via the REST APIs through unique resource ids.

For the media that is managed by DocumentDB, the `_media` property of the attachment will reference the media by its URI. DocumentDB will ensure to garbage collect the media when all of the outstanding references are dropped. DocumentDB automatically generates the attachment when you upload the new media and populates the `_media` to point to the newly added media. If you choose to store the media in a remote blob store managed by you (e.g. OneDrive, Azure Storage, DropBox etc), you can still use attachments to reference the media. In this case, you will create the attachment yourself and populate its `_media` property.

As with all other resources, attachments can be created, replaced, deleted, read or enumerated easily using either REST APIs or any of the client SDKs. As with documents, the read consistency level of attachments follows the consistency policy on the database account. This policy can be overridden on a per-request basis depending on data consistency requirements of your application. When querying for attachments, the read consistency follows the indexing mode set on the collection. For “consistent”, this follows the account’s consistency policy.

Users

A DocumentDB user represents a logical namespace for grouping permissions. A DocumentDB user may correspond to a user in an identity management system or a predefined application role. For DocumentDB, a user simply represents an abstraction to group a set of permissions under a database.

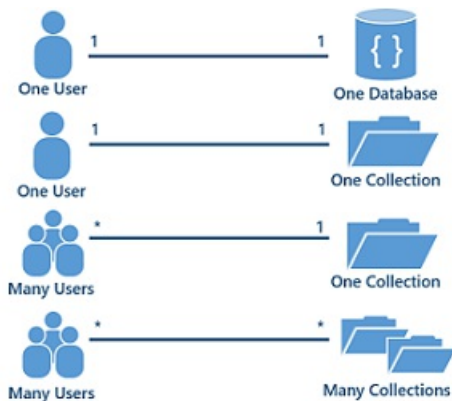
For implementing multi-tenancy in your application, you can create users in DocumentDB which corresponds to your actual users or the tenants of your application. You can then create permissions for a given user that

correspond to the access control over various collections, documents, attachments, etc.

As your applications need to scale with your user growth, you can adopt various ways to shard your data. You can model each of your users as follows:

- Each user maps to a database.
- Each user maps to a collection.
- Documents corresponding to multiple users go to a dedicated collection.
- Documents corresponding to multiple users go to a set of collections.

Regardless of the specific sharding strategy you choose, you can model your actual users as users in DocumentDB database and associate fine grained permissions to each user.



Sharding strategies and modeling users

Like all other resources, users in DocumentDB can be created, replaced, deleted, read or enumerated easily using either REST APIs or any of the client SDKs. DocumentDB always provides strong consistency for reading or querying the metadata of a user resource. It is worth pointing out that deleting a user automatically ensures that you cannot access any of the permissions contained within it. Even though the DocumentDB reclaims the quota of the permissions as part of the deleted user in the background, the deleted permissions is available instantly again for you to use.

Permissions

From an access control perspective, resources such as database accounts, databases, users and permission are considered *administrative* resources since these require administrative permissions. On the other hand, resources including the collections, documents, attachments, stored procedures, triggers, and UDFs are scoped under a given database and considered *application resources*. Corresponding to the two types of resources and the roles that access them (namely the administrator and user), the authorization model defines two types of *access keys*: *master key* and *resource key*. The master key is a part of the database account and is provided to the developer (or administrator) who is provisioning the database account. This master key has administrator semantics, in that it can be used to authorize access to both administrative and application resources. In contrast, a resource key is a granular access key that allows access to a *specific* application resource. Thus, it captures the relationship between the user of a database and the permissions the user has for a specific resource (e.g. collection, document, attachment, stored procedure, trigger, or UDF).

The only way to obtain a resource key is by creating a permission resource under a given user. Note that In order to create or retrieve a permission, a master key must be presented in the authorization header. A permission resource ties the resource, its access and the user. After creating a permission resource, the user only needs to present the associated resource key in order to gain access to the relevant resource. Hence, a resource key can be viewed as a logical and compact representation of the permission resource.

As with all other resources, permissions in DocumentDB can be created, replaced, deleted, read or enumerated easily using either REST APIs or any of the client SDKs. DocumentDB always provides strong consistency for

reading or querying the metadata of a permission.

Next steps

Learn more about working with resources by using HTTP commands in [RESTful interactions with DocumentDB resources](#).

Distribute data globally with DocumentDB

11/22/2016 • 4 min to read • [Edit on GitHub](#)

Contributors

Kirat Pandya • mimig • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

NOTE

Global distribution of DocumentDB databases is generally available and automatically enabled for any newly created DocumentDB accounts. We are working to enable global distribution on all existing accounts, but in the interim, if you want global distribution enabled on your account, please [contact support](#) and we'll enable it for you now.

Azure DocumentDB is designed to meet the needs of IoT applications consisting of millions of globally distributed devices and internet scale applications that deliver highly responsive experiences to users across the world. These database systems face the challenge of achieving low latency access to application data from multiple geographic regions with well-defined data consistency and availability guarantees. As a globally distributed database system, DocumentDB simplifies the global distribution of data by offering fully managed, multi-region database accounts that provide clear tradeoffs between consistency, availability and performance, all with corresponding guarantees. DocumentDB database accounts are offered with high availability, single digit ms latencies, multiple [well-defined consistency levels](#), transparent regional failover with multi-homing APIs, and the ability to elastically scale throughput and storage across the globe.

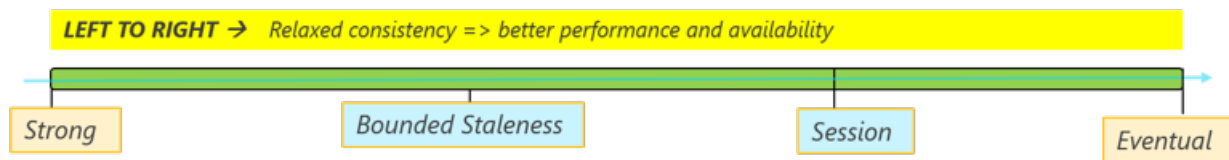
We recommend getting started by watching the following video, where Karthik Raman explains geo-distribution with Azure DocumentDB.



Configuring multi-region accounts

Configuring your DocumentDB account to scale across the globe can be done in less than a minute through the [Azure portal](#). All you need to do is select the right consistency level among several supported well-defined

consistency levels, and associate any number of Azure regions with your database account. DocumentDB consistency levels provide clear tradeoffs between specific consistency guarantee and performance.



DocumentDB offers multiple, well defined (relaxed) consistency models to choose from.

Selecting the right consistency level depends on data consistency guarantee your application needs. DocumentDB automatically replicates your data across all specified regions and guarantees the consistency that you have selected for your database account.

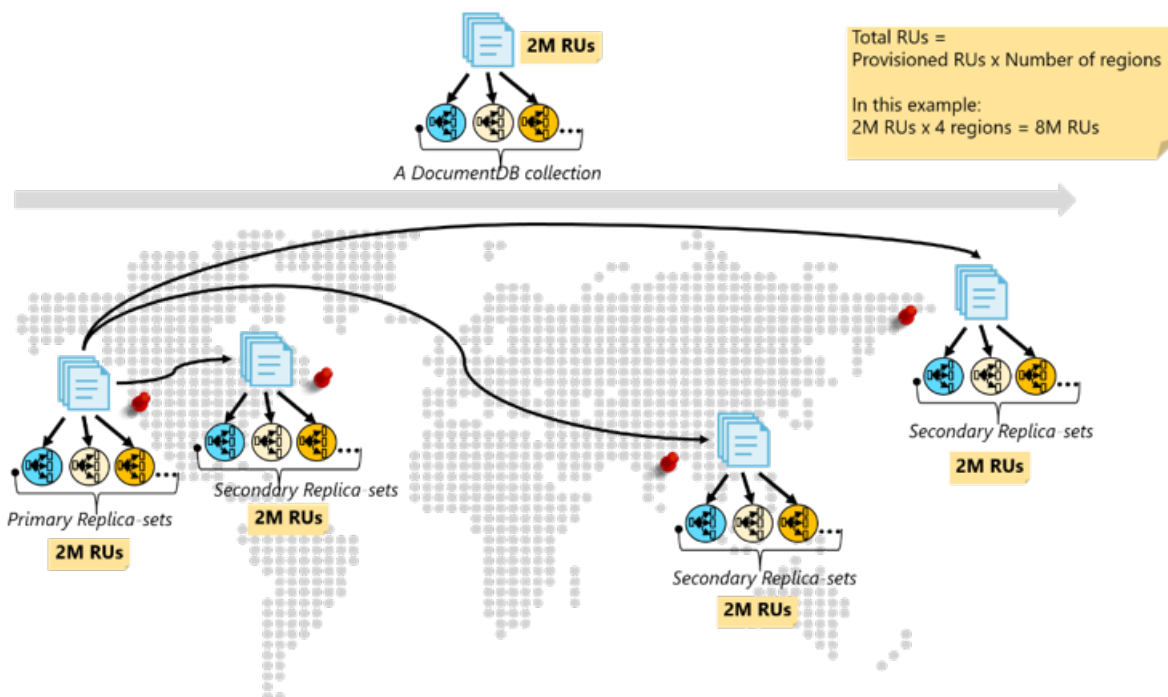
Using multi-region failover

Azure DocumentDB is able to transparently failover database accounts across multiple Azure regions – the new [multi-homing APIs](#) guarantee that your app can continue to use a logical endpoint and is uninterrupted by the failover. Failover is controlled by you, providing the flexibility to rehome your database account in the event any of range of possible failure conditions occur, including application, infrastructure, service or regional failures (real or simulated). In the event of a DocumentDB regional failure, the service will transparently fail over your database account and your application continues to access data without losing availability. While DocumentDB offers [99.99% availability SLAs](#), you can test your application's end to end availability properties by simulating a regional failure both, [programmatically](#) as well as through the Azure Portal.

Scaling across the planet

DocumentDB allows you to independently provision throughput and consume storage for each DocumentDB collection at any scale, globally across all the regions associated with your database account. A DocumentDB collection is automatically distributed globally and managed across all of the regions associated with your database account. Collections within your database account can be distributed across any of the Azure regions in which the [DocumentDB service is available](#).

The throughput purchased and storage consumed for each DocumentDB collection is automatically provisioned across all regions equally. This allows your application to seamlessly scale across the globe [paying only for the throughput and storage you are using within each hour](#). For instance, if you have provisioned 2 million RUs for a DocumentDB collection, then each of the regions associated with your database account gets 2 million RUs for that collection. This is illustrated below.



DocumentDB guarantees < 10 ms read and < 15 ms write latencies at P99. The read requests never span datacenter boundary to guarantee the [consistency requirements you have selected](#). The writes are always quorum committed locally before they are acknowledged to the clients. Each database account is configured with write region priority. The region designated with highest priority will act as the current write region for the account. All SDKs will transparently route database account writes to the current write region.

Finally, since DocumentDB is completely [schema-agnostic](#) - you never have to worry about managing/updating schemas or secondary indexes across multiple datacenters. Your [SQL queries](#) continue to work while your application and data models continue to evolve.

Enabling global distribution

You can decide to make your data locally or globally distributed by either associating one or more Azure regions with a DocumentDB database account. You can add or remove regions to your database account at any time. To enable global distribution by using the portal, see [How to perform DocumentDB global database replication using the Azure portal](#). To enable global distribution programmatically, see [Developing with multi-region DocumentDB accounts](#).

Next steps

Learn more about the distributing data globally with DocumentDB in the following articles:

- [Provisioning throughput and storage for a collection](#)
- [Multi-homing APIs via REST, .NET, Java, Python, and Node SDKs](#)
- [Consistency Levels in DocumentDB](#)
- [Availability SLAs](#)
- [Managing database account](#)

Common DocumentDB use cases

11/22/2016 • 8 min to read • [Edit on GitHub](#)

Contributors

[Han Wong](#) • [mimig](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [arramac](#) • [Dene Hager](#)

This article provides an overview of several common use cases for DocumentDB. The recommendations in this article serve as a starting point as you develop your application with DocumentDB.

After reading this article, you'll be able to answer the following questions:

- What are the common use cases for DocumentDB?
- What are the benefits of using DocumentDB for web and mobile applications?
- What are the benefits of using DocumentDB for retail applications?
- What are the benefits of using DocumentDB as a data store for Internet of Things (IoT) systems?
- What are the benefits of using DocumentDB as an event log store?

Common use cases for DocumentDB

Azure DocumentDB is a general purpose NoSQL database that is used in a wide range of applications and use cases. It is a good choice for any application that needs low order-of-millisecond response times, and needs to scale rapidly. The following are some attributes of DocumentDB that make it well-suited for high-performance applications.

- DocumentDB natively partitions your data for high availability and scalability.
- DocumentDB's has SSD backed storage with low-latency order-of-millisecond response times.
- DocumentDB's support for consistency levels like eventual, session and bounded-staleness allows for low cost-to performance-ratio.
- DocumentDB has a flexible data-friendly pricing model that meters storage and throughput independently.
- DocumentDB's reserved throughput model allows you to think in terms of number of reads/writes instead of CPU/memory/IOPs of the underlying hardware.
- DocumentDB's design lets you scale to massive request volumes in the order of billions of requests per day.

These attributes are particularly beneficial when it comes to web, mobile, gaming and IoT applications that need low response times and need to handle massive amounts of reads and writes.

Web and mobile applications

DocumentDB is commonly used within web and mobile applications, and is particularly well suited for modeling social interactions, integrating with third-party services, and for building rich personalized experiences.

Social Applications

A common use case for DocumentDB is to store and query user generated content (UGC) for web and mobile applications, particularly social media applications. Some examples of UGC are chat sessions, tweets, blog posts, ratings, and comments. Often, the UGC in social media applications is a blend of free form text, properties, tags and relationships that are not bounded by rigid structure. Content such as chats, comments, and posts can be stored in DocumentDB without requiring transformations or complex object to relational mapping layers. Data properties can be added or modified easily to match requirements as developers iterate over the application code, thus promoting rapid development.

Applications that integrate with third-party social networks must respond to changing schemas from these networks. As data is automatically indexed by default in DocumentDB, data is ready to be queried at any time. Hence, these applications have the flexibility to retrieve projections as per their respective needs.

Many of the social applications run at global scale and can exhibit unpredictable usage patterns. Flexibility in scaling the data store is essential as the application layer scales to match usage demand. You can scale out by adding additional data partitions under a DocumentDB account. In addition, you can also create additional DocumentDB accounts across multiple regions. For DocumentDB service region availability, see [Azure Regions](#).

Personalization

Nowadays, modern applications come with complex views and experiences. These are typically dynamic, catering to user preferences or moods and branding needs. Hence, applications need to be able to retrieve personalized settings effectively in order to render UI elements and experiences quickly.

JSON is an effective format to represent UI layout data as it is not only lightweight, but also can be easily interpreted by JavaScript. DocumentDB offers tunable consistency levels that allow fast reads with low latency writes. Hence, storing UI layout data including personalized settings as JSON documents in DocumentDB is an effective means to get this data across the wire.

Retail applications

DocumentDB is commonly used in the retail industry for storing catalog data. Catalog data usage scenarios involve storing and querying a set of attributes for entities such as people, places and products. Some examples of catalog data are user accounts, product catalogs, device registries for IoT, and bill of materials systems. Attributes for this data may vary and can change over time to fit application requirements.

Consider an example of a product catalog for an automotive parts supplier. Every part may have its own attributes in addition to the common attributes that all parts share. Furthermore, attributes for a specific part can change the following year when a new model is released. As a JSON document store, DocumentDB supports flexible schemas and allows you to represent data with nested properties, and thus it is well suited for storing product catalog data.

IoT and Telematics

IoT use cases commonly share some patterns in how they ingest, process and store data. First, these systems allow for data intake that can ingest bursts of data from device sensors of various locales. Next, these systems process and analyze streaming data to derive real time insights. And last but not least, most if not all data will eventually land in a data store for adhoc querying and offline analytics.

Microsoft Azure offers rich services that can be leveraged for IoT use cases. Azure IoT services are a set of services including Azure Event Hubs, Azure DocumentDB, Azure Stream Analytics, Azure Notification Hub, Azure Machine Learning, Azure HDInsight, and PowerBI.

Bursts of data can be ingested by Azure Event Hubs as it offers high throughput data ingestion with low latency. Data ingested that needs to be processed for real time insight can be funneled to Azure Stream Analytics for real time analytics. Data can be loaded into DocumentDB for adhoc querying. Once the data is loaded into DocumentDB, the data is ready to be queried. The data in DocumentDB can be used as reference data as part of real time analytics. In addition, data can further be refined and processed by connecting DocumentDB data to HDInsight for Pig, Hive or Map/Reduce jobs. Refined data is then loaded back to DocumentDB for reporting.

For a sample IoT solution using DocumentDB, EventHubs and Storm, see the [hdinsight-storm-examples repository on GitHub](#).

For more information on Azure offerings for IoT, see [Create the Internet of Your Things](#).

Logging and Time-series data

Application logging is often emitted in large volumes and may have varying attributes based on the deployed application version or the component logging events. Log data is not bounded by complex relationships or rigid structures. Increasingly, log data is persisted in JSON format since JSON is lightweight and easy for humans to read.

There are typically two major use cases related to event log data. The first use case is to perform ad-hoc queries over a subset of data for troubleshooting. During troubleshooting, a subset of data is first retrieved from the logs, typically by time series. Then, a drill-down is performed by filtering the dataset with error levels or error messages. This is where storing event logs in DocumentDB is an advantage. Log data stored in DocumentDB is automatically indexed by default, and thus it is ready to be queried at any time. In addition, log data can be persisted across data partitions as a time-series. Older logs can be rolled out to cold storage per your retention policy.

The second use case involves long running data analytics jobs performed offline over a large volume of log data. Examples of this use case include server availability analysis, application error analysis, and clickstream data analysis. Typically, Hadoop is used to perform these types of analyses. With the Hadoop Connector for DocumentDB, DocumentDB databases function as data sources and sinks for Pig, Hive and Map/Reduce jobs. For details on the Hadoop Connector for DocumentDB, see [Run a Hadoop job with DocumentDB and HDInsight](#).

Gaming

The database tier is a crucial component of gaming applications. Modern games perform graphical processing on mobile/console clients, but rely on the cloud to deliver customized and personalized content like in-game stats, social media integration, and high-score leaderboards. Games require extremely low latencies for reads and writes to provide an engaging in-game experience, and the database tier needs to handle highs and lows in request rates during new game launches and feature updates.

DocumentDB is used by massive-scale games like [The Walking Dead: No Man's Land](#) by [Next Games](#), and [Halo 5: Guardians](#). In both use cases, the key advantages of DocumentDB were the following:

- DocumentDB allows performance to be scaled up or down elastically. This allows games to handle updating profile and stats from dozens to millions of simultaneous gamers by making a single API call.
- DocumentDB supports millisecond reads and writes to help avoid any lags during game play.
- DocumentDB's automatic indexing allows for filtering against multiple different properties in real-time, e.g. locate players by their internal player IDs, or their GameCenter, Facebook, Google IDs, or query based on player membership in a guild. This is possible without building complex indexing or sharding infrastructure.
- Social features including in-game chat messages, player guild memberships, challenges completed, high-score leaderboards, and social graphs are easier to implement with a flexible schema.
- DocumentDB as a managed platform-as-a-service (PaaS) required minimal setup and management work to allow for rapid iteration, and reduce time to market.

Next steps

To get started with DocumentDB, you can create an [account](#) and then follow our [learning path](#) to learn about DocumentDB and find the information you need.

Or, if you'd like to read more about customers using DocumentDB, the following customer stories are available:

- [Affinio](#). Affinio switches from AWS to Azure DocumentDB to harness social data at scale.
- [Next Games](#). The Walking Dead: No Man's Land game soars to #1 supported by Azure DocumentDB.
- [Halo](#). How Halo 5 implemented social gameplay using Azure DocumentDB.
- [Cortana Analytics Gallery](#). Cortana Analytics Gallery - a scalable community site built on Azure DocumentDB.
- [Breeze](#). Leading Integrator Gives Multinational Firms Global Insight in Minutes with Flexible Cloud Technologies.
- [News Republic](#). Adding intelligence to the news to provide information with purpose for engaged citizens.
- [SGS International](#). For consistent color across the globe, major brands turn to SGS. And SGS turns to Azure.

- [Telenor](#). Global leader Telenor uses the cloud to move with the speed of a startup.
- [XOMNI](#). The store of the future runs on speedy search and the easy flow of data.
- [Nucleo](#). Azure-based software platform breaks down barriers between businesses and customers
- [Weka](#). Weka Smart Fridge improves vaccine management so more people can be protected against diseases
- [Orange Tribes](#). There's more to that food app than meets the eye, or the mouth.
- [Real Madrid](#). Real Madrid brings the stadium closer to 450 million fans around the globe, with the Microsoft Cloud.
- [Tuku](#). TUKU makes car buying fun with help from Azure services

Going social with DocumentDB

11/15/2016 • 11 min to read • [Edit on GitHub](#)

Contributors

Matias Quaranta • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • mimig • Gary Ericson

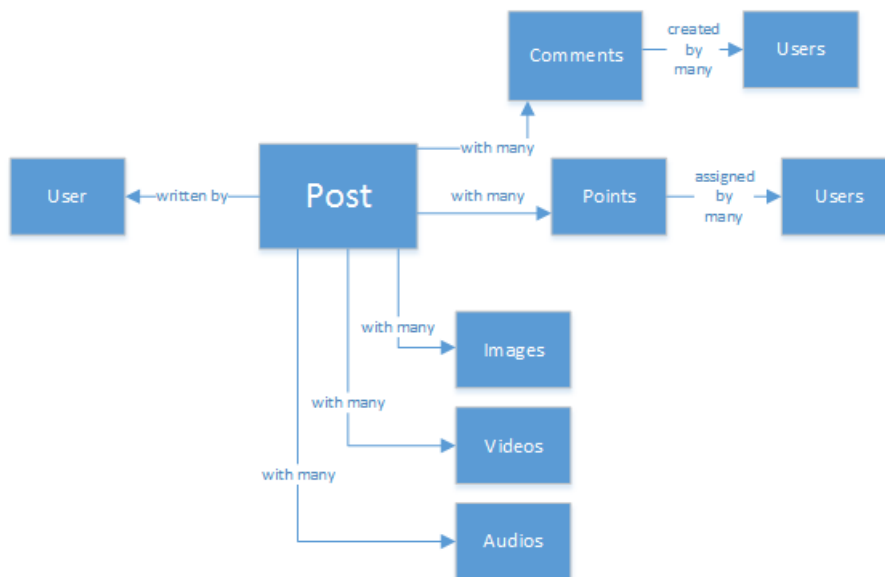
Living in a massively-interconnected society means that, at some point in life, you become part of a **social network**. We use social networks to keep in touch with friends, colleagues, family, or sometimes to share our passion with people with common interests.

As engineers or developers, we might have wondered how do these networks store and interconnect our data, or might have even been tasked to create or architect a new social network for a specific niche market yourselves. That's when the big question arises: How is all this data stored?

Let's suppose that we are creating a new and shiny social network, where our users can post articles with related media like, pictures, videos, or even music. Users can comment on posts and give points for ratings. There will be a feed of posts that users will see and be able to interact with on the main website landing page. This doesn't sound really complex (at first), but for the sake of simplicity, let's stop there (we could delve into custom user feeds affected by relationships, but it exceeds the goal of this article).

So, how do we store this and where?

Many of you might have experience on SQL databases or at least have notion of [relational modeling of data](#) and you might be tempted to start drawing something like this:



A perfectly normalized and pretty data structure... that doesn't scale.

Don't get me wrong, I've worked with SQL databases all my life, they are great, but like every pattern, practice and software platform, it's not perfect for every scenario.

Why isn't SQL the best choice in this scenario? Let's look at the structure of a single post, if I wanted to show that post in a website or application, I'd have to do a query with... 8 table joins (!) just to show one single post, now, picture a stream of posts that dynamically load and appear on the screen and you might see where I am going.

We could, of course, use a humongous SQL instance with enough power to solve thousands of queries with these many joins to serve our content, but truly, why would we when a simpler solution exists?

The NoSQL road

There are special graph databases that can [run on Azure](#) but they are not inexpensive and require IaaS services (Infrastructure-as-a-Service, Virtual Machines mainly) and maintenance. I'm going to aim this article at a lower cost solution that will work for most scenarios, running on Azure's NoSQL database [DocumentDB](#). Using a [NoSQL](#) approach, storing data in JSON format and applying [denormalization](#), our previously complicated post can be transformed into a single [Document](#):

```
{
  "id": "ew12-res2-234e-544f",
  "title": "post title",
  "date": "2016-01-01",
  "body": "this is an awesome post stored on NoSQL",
  "createdBy": "User",
  "images": ["http://myfirstimage.png", "http://mysecondimage.png"],
  "videos": [
    { "url": "http://myfirstvideo.mp4", "title": "The first video" },
    { "url": "http://mysecondvideo.mp4", "title": "The second video" }
  ],
  "audios": [
    { "url": "http://myfirstaudio.mp3", "title": "The first audio" },
    { "url": "http://mysecondaudio.mp3", "title": "The second audio" }
  ]
}
```

And it can be obtained with a single query, and with no joins. This is much more simple and straightforward, and, budget-wise, it requires fewer resources to achieve a better result.

Azure DocumentDB makes sure that all the properties are indexed with its [automatic indexing](#), which can even be [customized](#). The schema-free approach lets us store Documents with different and dynamic structures, maybe tomorrow we want posts to have a list of categories or hashtags associated with them, DocumentDB will handle the new Documents with the added attributes with no extra work required by us.

Comments on a post can be treated as just other posts with a parent property (this simplifies our object mapping).

```
{
  "id": "1234-asd3-54ts-199a",
  "title": "Awesome post!",
  "date": "2016-01-02",
  "createdBy": "User2",
  "parent": "ew12-res2-234e-544f"
}

{
  "id": "asd2-fee4-23gc-jh67",
  "title": "Ditto!",
  "date": "2016-01-03",
  "createdBy": "User3",
  "parent": "ew12-res2-234e-544f"
}
```

And all social interactions can be stored on a separate object as counters:

```
{
  "id": "dfe3-thf5-232s-dse4",
  "post": "ew12-res2-234e-544f",
  "comments": 2,
  "likes": 10,
  "points": 200
}
```

Creating feeds is just a matter of creating documents that can hold a list of post ids with a given relevance order:

```
[
  {"relevance":9, "post":"ew12-res2-234e-544f"},
  {"relevance":8, "post":"fer7-mnb6-fgh9-2344"},
  {"relevance":7, "post":"w34r-qeg6-ref6-8565"}
]
```

We could have a “latest” stream with posts ordered by creation date, a “hottest” stream with those posts with more likes in the last 24 hours, we could even implement a custom stream for each user based on logic like followers and interests, and it would still be a list of posts. It’s a matter of how to build these lists, but the reading performance remains unhindered. Once we acquire one of these lists, we issue a single query to DocumentDB using the [IN operator](#) to obtain pages of posts at a time.

The feed streams could be built using [Azure App Services’](#) background processes: [Webjobs](#). Once a post is created, background processing can be triggered by using [Azure Storage Queues](#) and Webjobs triggered using the [Azure Webjobs SDK](#), implementing the post propagation inside streams based on our own custom logic.

Points and likes over a post can be processed in a deferred manner using this same technique to create an eventually consistent environment.

Followers are trickier. DocumentDB has a document size limit of 512Kb, so you may think about storing followers as a document with this structure:

```
{
  "id":"234d-sd23-rrf2-552d",
  "followersOf": "dse4-qwe2-ert4-aad2",
  "followers":[
    "ewr5-232d-tyrg-iuo2",
    "qejh-2345-sdf1-ytg5",
    //...
    "uie0-4tyg-3456-rwjh"
  ]
}
```

This might work for a user with a few thousands followers, but if some celebrity joins our ranks, this approach will eventually hit the document size cap.

To solve this, we can use a mixed approach. As part of the User Statistics document we can store the number of followers:

```
{
  "id":"234d-sd23-rrf2-552d",
  "user": "dse4-qwe2-ert4-aad2",
  "followers":55230,
  "totalPosts":452,
  "totalPoints":11342
}
```

And the actual graph of followers can be stored on Azure Storage Tables using an [Extension](#) that allows for simple “A-follows-B” storage and retrieval. This way we can delegate the retrieval process of the exact followers list (when we need it) to Azure Storage Tables but for a quick numbers lookup, we keep using DocumentDB.

The “Ladder” pattern and data duplication

As you might have noticed in the JSON document that references a post, there are multiple occurrences of a user. And you’d have guessed right, this means that the information that represents a user, given this denormalization, might be present in more than one place.

In order to allow for faster queries, we incur data duplication. The problem with this side-effect is that if by some action, a user's data changes, we need to find all the activities he ever did and update them all. Doesn't sound very practical, right?

Graph databases solve it in their own way, we are going to solve it by identifying the Key attributes of a user that we show in our application for each activity. If we visually show a post in our application and show just the creator's name and picture, why store all of the user's data in the "createdBy" attribute? If for each comment we just show the user's picture, we don't really need the rest of his information. That's where something I call the "Ladder pattern" comes into play.

Let's take user information as an example:

```
{
  "id": "dse4-qwe2-ert4-aad2",
  "name": "John",
  "surname": "Doe",
  "address": "742 Evergreen Terrace",
  "birthday": "1983-05-07",
  "email": "john@doe.com",
  "twitterHandle": "@john",
  "username": "johndoe",
  "password": "some_encrypted_phrase",
  "totalPoints": 100,
  "totalPosts": 24
}
```

By looking at this information, we can quickly detect which is critical information and which isn't, thus creating a "Ladder":

id	name												
		totalPoints	totalPosts	surname	email	twitterHandle	following						
								username	password	phone	address	birthday	

The smallest step is called a UserChunk, the minimal piece of information that identifies a user and it's used for data duplication. By reducing the size of the duplicated data to only the information we will "show", we reduce the possibility of massive updates.

The middle step is called the user, it's the full data that will be used on most performance-dependent queries on DocumentDB, the most accessed and critical. It includes the information represented by a UserChunk.

The largest is the Extended User. It includes all the critical user information plus other data that doesn't really require to be read quickly or it's usage is eventual (like the login process). This data can be stored outside of DocumentDB, in Azure SQL Database or Azure Storage Tables.

Why would we split the user and even store this information in different places? Because storage space in DocumentDB is [not infinite](#) and from a performance point of view, the bigger the documents, the costlier the queries. Keep documents slim, with the right information to do all your performance-dependent queries for your social network, and store the other extra information for eventual scenarios like, full profile edits, logins, even data mining for usage analytics and Big Data initiatives. We really don't care if the data gathering for data mining is slower because it's running on Azure SQL Database, we do have concern though that our users have a fast and slim experience. A user, stored on DocumentDB, would look like this:

```
{
  "id": "dse4-qwe2-ert4-aad2",
  "name": "John",
  "surname": "Doe",
  "username": "johndoe",
  "email": "john@doe.com",
  "twitterHandle": "@john"
}
```

And a Post would look like:

```
{
  "id": "1234-asd3-54ts-199a",
  "title": "Awesome post!",
  "date": "2016-01-02",
  "createdBy": {
    "id": "dse4-qwe2-ert4-aad2",
    "username": "johndoe"
  }
}
```

And when an edit arises where one of the attributes of the chunk is affected, it's easy to find the affected documents by using queries that point to the indexed attributes (`SELECT * FROM posts p WHERE p.createdBy.id == "edited_user_id"`) and then updating the chunks.

The search box

Users will generate, luckily, a lot of content. And we should be able to provide the ability to search and find content that might not be directly in their content streams, maybe because we don't follow the creators, or maybe we are just trying to find that old post we did 6 months ago.

Thankfully, and because we are using Azure DocumentDB, we can easily implement a search engine using [Azure Search](#) in a couple of minutes and without typing a single line of code (other than obviously, the search process and UI).

Why is this so easy?

Azure Search implements what they call [Indexers](#), background processes that hook in your data repositories and automatically add, update or remove your objects in the indexes. They support an [Azure SQL Database indexers](#), [Azure Blobs indexers](#) and thankfully, [Azure DocumentDB indexers](#). The transition of information from DocumentDB to Azure Search is straightforward, as both store information in JSON format, we just need to [create our Index](#) and map which attributes from our Documents we want indexed and that's it, in a matter of minutes (depends on the size of our data), all our content will be available to be searched upon, by the best Search-as-a-Service solution in cloud infrastructure.

For more information about Azure Search, you can visit the [Hitchhiker's Guide to Search](#).

The underlying knowledge

After storing all this content that grows and grows every day, we might find ourselves thinking: What can I do with all this stream of information from my users?

The answer is straightforward: Put it to work and learn from it.

But, what can we learn? A few easy examples include [sentiment analysis](#), content recommendations based on a user's preferences or even an automated content moderator that ensures that all the content published by our social network is safe for the family.

Now that I got you hooked, you'll probably think you need some PhD in math science to extract these patterns and

information out of simple databases and files, but you'd be wrong.

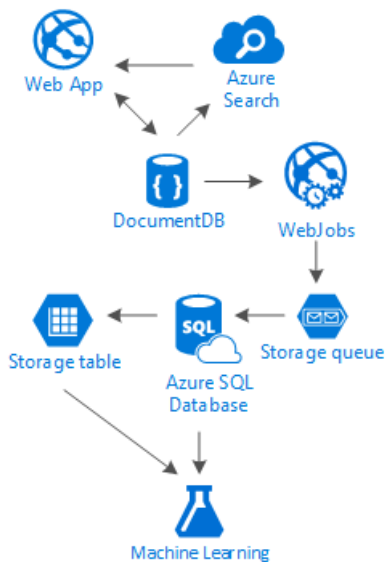
[Azure Machine Learning](#), part of the [Cortana Intelligence Suite](#), is the a fully managed cloud service that lets you create workflows using algorithms in a simple drag-and-drop interface, code your own algorithms in [R](#) or use some of the already-built and ready to use APIs such as: [Text Analytics](#), [Content Moderator](#) or [Recommendations](#).

To achieve any of these Machine Learning scenarios, we can use [Azure Data Lake](#) to ingest the information from different sources, and use [U-SQL](#) to process the information and generate an output that can be processed by Azure Machine Learning.

Another available option is to use [Microsoft Cognitive Services](#) to analyze our users content; not only can we understand them better (through analyzing what they write with [Text Analytics API](#)) , but we could also detect unwanted or mature content and act accordingly with [Computer Vision API](#). Cognitive Services include a lot of out-of-the-box solutions that don't require any kind of Machine Learning knowledge to use.

Conclusion

This article tries to shed some light into the alternatives of creating social networks completely on Azure with low-cost services and providing great results by encouraging the use of a multi-layered storage solution and data distribution called "Ladder".



The truth is that there is no silver bullet for this kind of scenarios, it's the synergy created by the combination of great services that allow us to build great experiences: the speed and freedom of Azure DocumentDB to provide a great social application, the intelligence behind a first-class search solution like Azure Search, the flexibility of Azure App Services to host not even language-agnostic applications but powerful background processes and the expandable Azure Storage and Azure SQL Database for storing massive amounts of data and the analytic power of Azure Machine Learning to create knowledge and intelligence that can provide feedback to our processes and help us deliver the right content to the right users.

Next steps

Learn more about data modeling by reading the [Modeling data in DocumentDB](#) article. If you're interested in other use cases for DocumentDB, see [Common DocumentDB use cases](#).

Or learn more about DocumentDB by following the [DocumentDB Learning Path](#).

NoSQL tutorial: Build a DocumentDB C# console application

11/22/2016 • 15 min to read • [Edit on GitHub](#)

Contributors

[Andrew Hoh](#) • [mimig](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [Carolyn Gronlund](#) • [Michiel Staessen](#) • [arramac](#) • [v-aljenk](#) • [Ross McAllister](#) • [Dene Hager](#)

Welcome to the NoSQL tutorial for the Azure 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](#). Jump to the [Get the complete solution section](#) 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

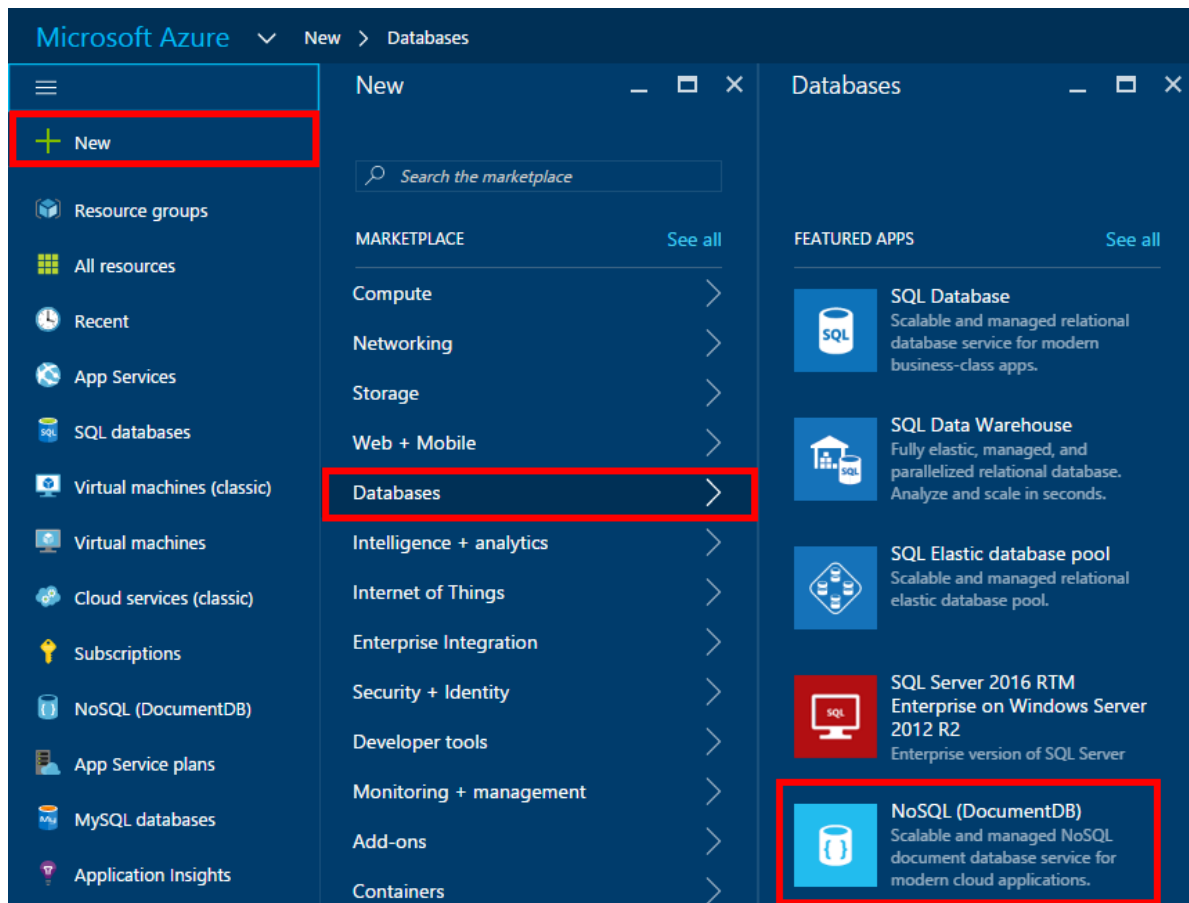
Please make sure you have the following:

- An active Azure account. If you don't have one, you can sign up for a [free account](#).
 - Alternatively, you can use the [Azure DocumentDB Emulator](#) for this tutorial.
- [Visual Studio 2013 / Visual Studio 2015](#).
- .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](#). If you are using the DocumentDB Emulator, please follow the steps at [Azure DocumentDB Emulator](#) to setup the emulator and skip ahead to [Setup your Visual Studio Solution](#).

1. Sign in to the [Azure portal](#).
2. In the Jumpbar, click **New**, click **Databases**, and then click **NoSQL (DocumentDB)**.



3. In the **New** account blade, specify the desired configuration for the DocumentDB account.

NoSQL (DocumentDB)
New account

* ID
contosoacct ✓
documents.azure.com

NoSQL API ⓘ
DocumentDB MongoDB

* Subscription
Visual Studio Ultimate with MSDN ▼

* Resource Group ⓘ
☒ Create new ☐ Use existing
contosoacct ✓

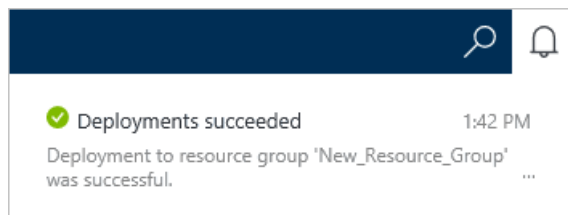
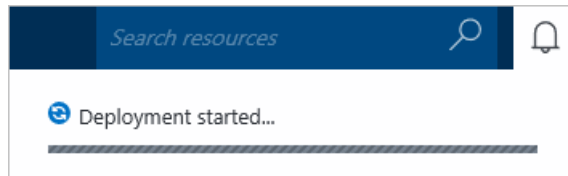
* Location
West US ▼

☐ Pin to dashboard

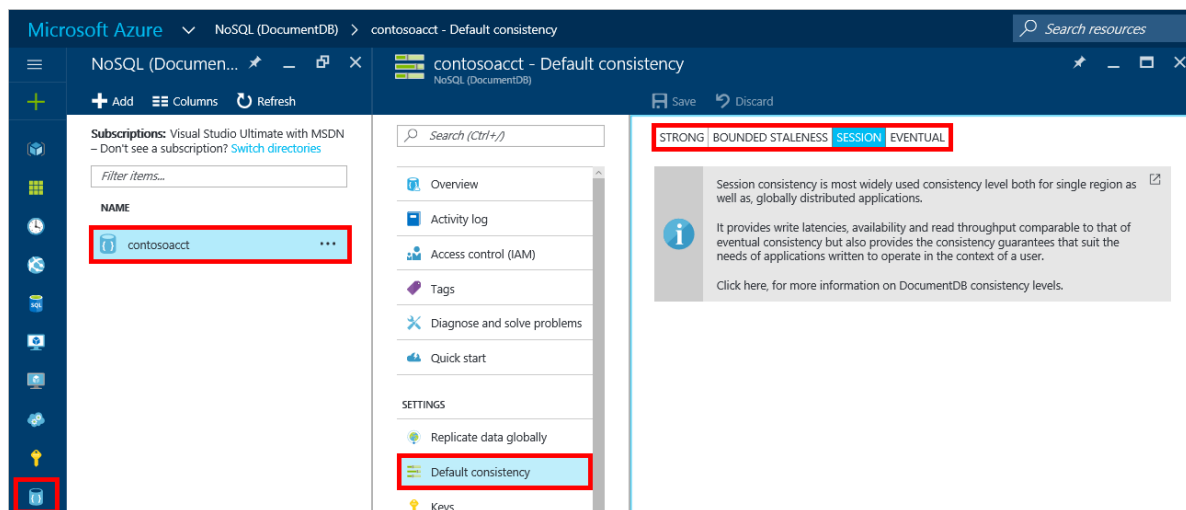
Create [Automation options](#)

- In the ID box, enter a name to identify the DocumentDB account. When the ID is validated, a green check mark appears in the ID box. The ID value becomes the host name within the URI. The ID may contain only lowercase letters, numbers, and the '-' character, and must be between 3 and 50 characters. Note that *documents.azure.com* is appended to the endpoint name you choose, the result of which becomes your DocumentDB account endpoint.

- In the **NoSQL API** box, select **DocumentDB**.
 - For **Subscription**, select the Azure subscription that you want to use for the DocumentDB account. If your account has only one subscription, that account is selected by default.
 - In **Resource Group**, select or create a resource group for your DocumentDB account. By default, a new resource group is created. For more information, see [Using the Azure portal to manage your Azure resources](#).
 - Use **Location** to specify the geographic location in which to host your DocumentDB account.
4. Once the new DocumentDB account options are configured, click **Create**. To check the status of the deployment, check the Notifications hub.



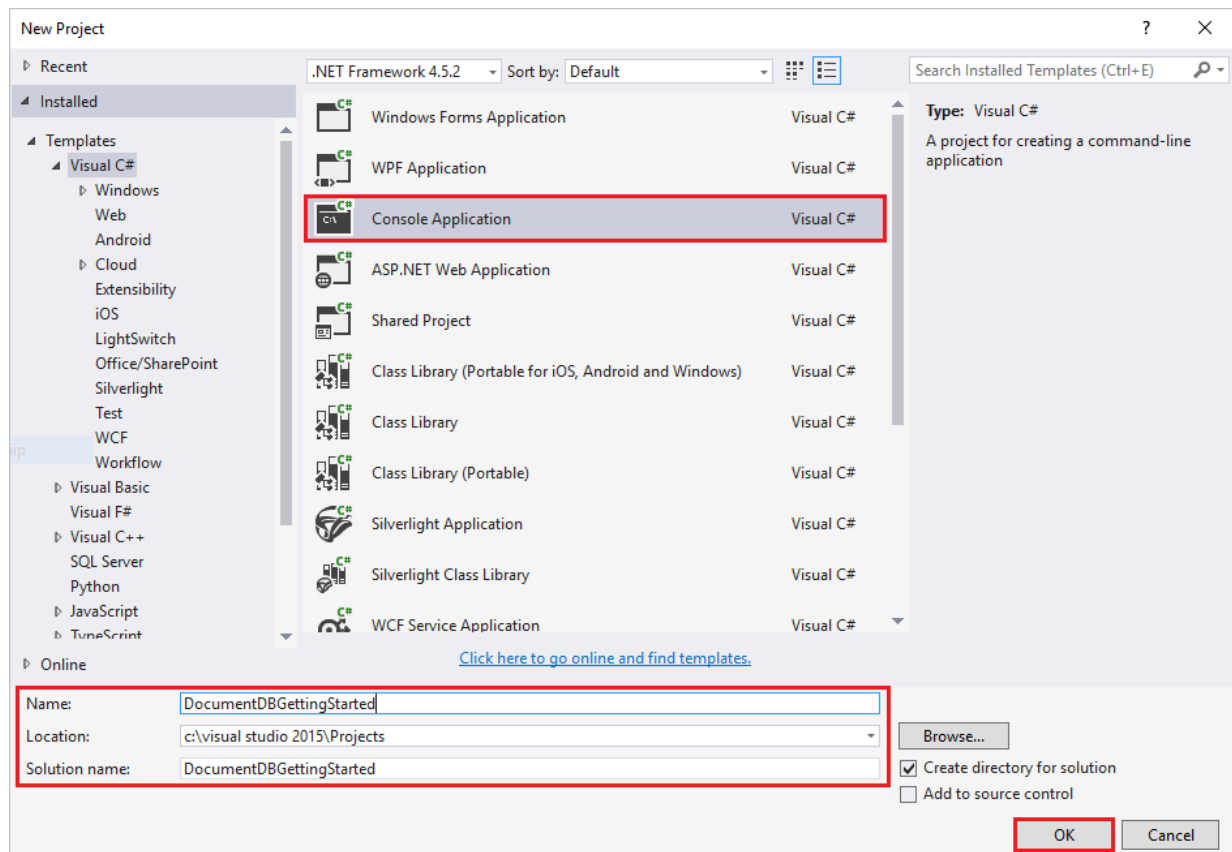
5. After the DocumentDB account is created, it is ready for use with the default settings. To review the default settings, click the **NoSQL (DocumentDB)** icon on the Jumpbar, click your new account, and then click **Default Consistency** in the Resource Menu.



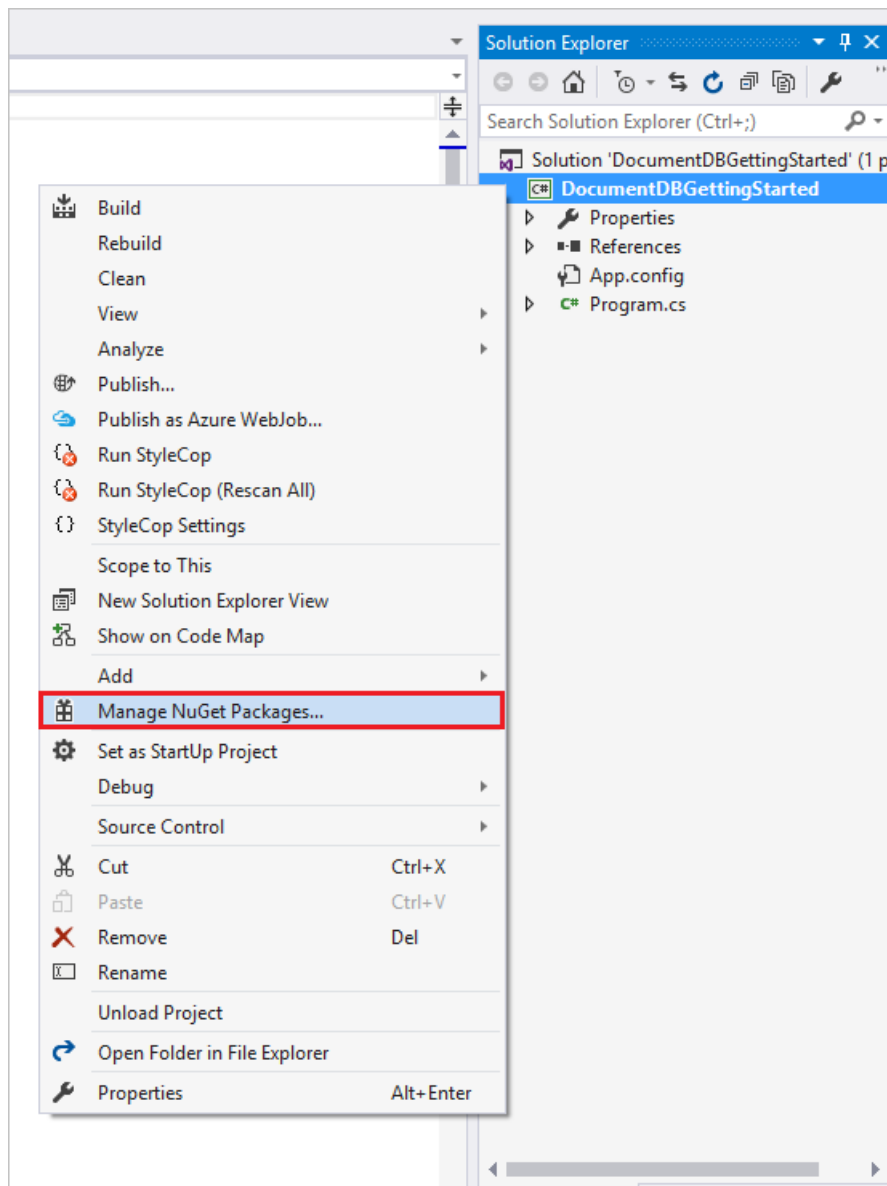
The default consistency of the DocumentDB account is set to **Session**. You can adjust the default consistency by clicking **Default Consistency** in the resource menu. To learn more about the consistency levels offered by DocumentDB, see [Consistency levels in DocumentDB](#).

Step 2: Setup your Visual Studio solution

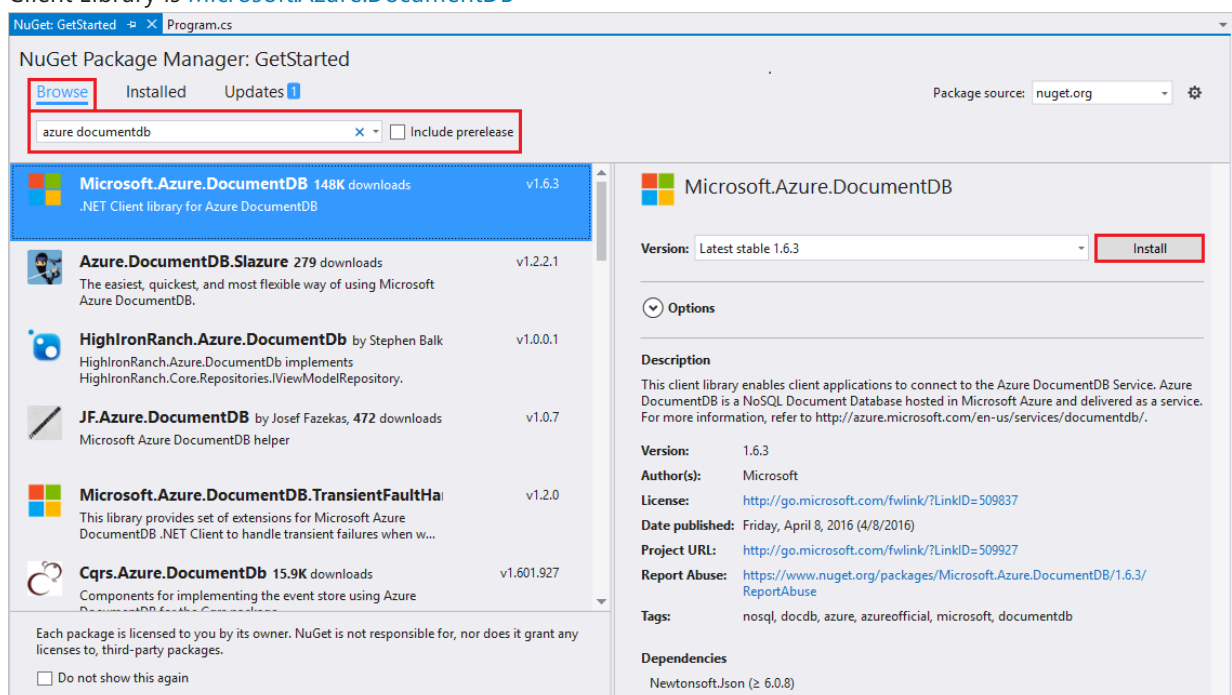
1. Open **Visual Studio 2015** on your computer.
2. On the **File** menu, select **New**, and then choose **Project**.
3. In the **New Project** dialog, select **Templates / Visual C# / Console Application**, name your project, and then click **OK**.



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](#)



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](#).

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;
```

IMPORTANT

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

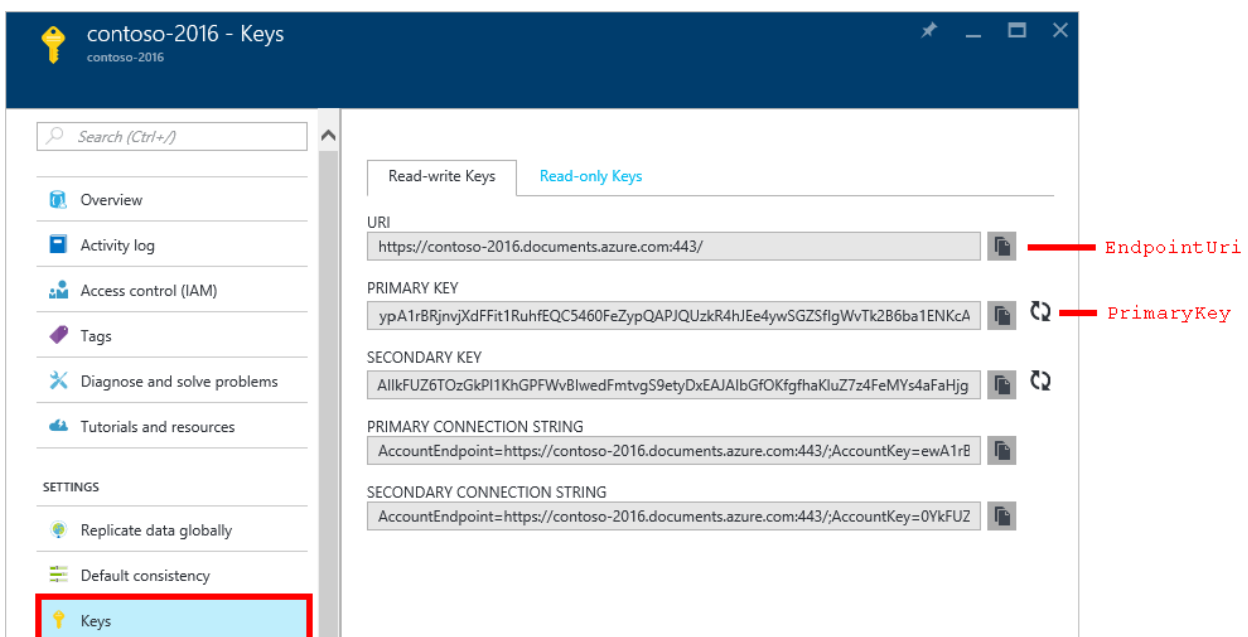
Now, add these two constants and your *client* variable underneath your public class *Program*.

```
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](#) 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, and then click **Keys**.

Copy the URI from the portal and paste it into `<your endpoint URI>` in the program.cs file. Then copy the PRIMARY KEY from the portal and paste it into `<your key>`.



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](#) can be created by using the [CreateDatabaseAsync](#) 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_oa");
}
```

Press **F5** to run your application.

Congratulations! You have successfully created a DocumentDB database.

Step 5: Create a collection

WARNING

CreateDocumentCollectionAsync will create a new collection with reserved throughput, which has pricing implications. For more details, please visit our [pricing page](#).

A [collection](#) can be created by using the [CreateDocumentCollectionAsync](#) 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),
                collectionInfo,
                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_oa*.

```
this.client = new DocumentClient(new Uri(EndpointUri), PrimaryKey);

await this.CreateDatabaseIfNotExists("FamilyDB_oa");

// ADD THIS PART TO YOUR CODE
await this.CreateDocumentCollectionIfNotExists("FamilyDB_oa", "FamilyCollection_oa");
```

Press F5 to run your application.

Congratulations! You have successfully created a DocumentDB document collection.

Step 6: Create JSON documents

A [document](#) can be created by using the [CreateDocumentAsync](#) 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](#).

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_oa");

await this.CreateDocumentCollectionIfNotExists("FamilyDB_oa", "FamilyCollection_oa");

// 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_oa", "FamilyCollection_oa", andersenFamily);

Family wakefieldFamily = new Family
{
    Id = "Wakefield.7",
    LastName = "Wakefield",
    Parents = new Parent[]
```

```

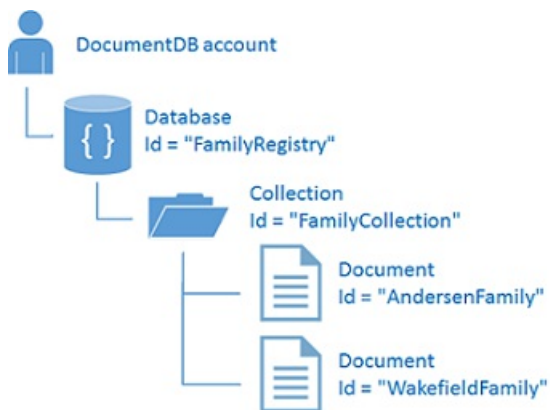
Parents = new Parent[]
{
    new Parent { FamilyName = "Wakefield", FirstName = "Robin" },
    new Parent { FamilyName = "Miller", FirstName = "Ben" }
},
Children = new Child[]
{
    new Child
    {
        FamilyName = "Merriam",
        FirstName = "Jesse",
        Gender = "female",
        Grade = 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_oa", "FamilyCollection_oa", wakefieldFamily);

```

Press **F5** to run your application.

Congratulations! You have successfully created two DocumentDB documents.



Step 7: Query DocumentDB resources

DocumentDB supports rich [queries](#) 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.

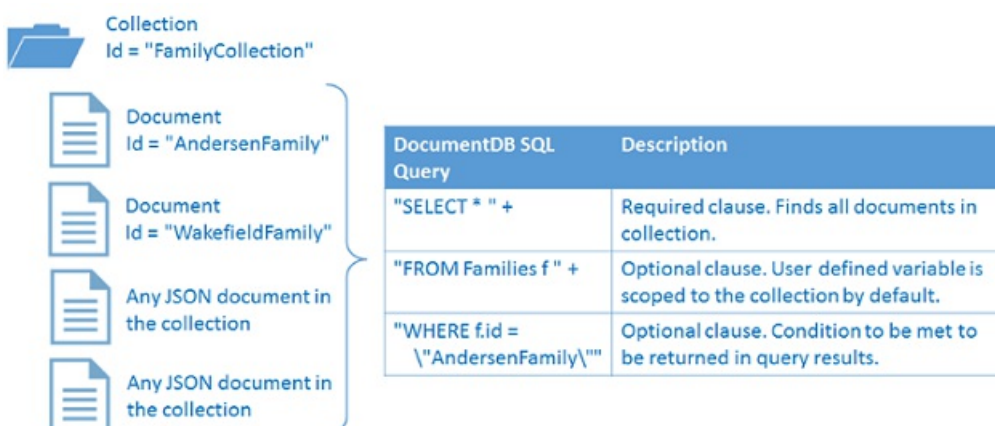
```
await this.CreateFamilyDocumentIfNotExists("FamilyDB_oa", "FamilyCollection_oa", wakefieldFamily);

// ADD THIS PART TO YOUR CODE
this.ExecuteSimpleQuery("FamilyDB_oa", "FamilyCollection_oa");
```

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.



The **FROM** 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;
    }
}
```

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_oa", "FamilyCollection_oa", wakefieldFamily);

this.ExecuteSimpleQuery("FamilyDB_oa", "FamilyCollection_oa");

// ADD THIS PART TO YOUR CODE
// Update the Grade of the Andersen Family child
andersenFamily.Children[0].Grade = 6;

await this.ReplaceFamilyDocument("FamilyDB_oa", "FamilyCollection_oa", "Andersen.1", andersenFamily);

this.ExecuteSimpleQuery("FamilyDB_oa", "FamilyCollection_oa");
```

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;
    }
}
```

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

```
await this.ReplaceFamilyDocument("FamilyDB_oa", "FamilyCollection_oa", "Andersen.1", andersenFamily);

this.ExecuteSimpleQuery("FamilyDB_oa", "FamilyCollection_oa");

// ADD THIS PART TO CODE
await this.DeleteFamilyDocument("FamilyDB_oa", "FamilyCollection_oa", "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_oa", "FamilyCollection_oa");

await this.DeleteFamilyDocument("FamilyDB_oa", "FamilyCollection_oa", "Andersen.1");

// ADD THIS PART TO CODE
// Clean up/delete the database
await this.client.DeleteDatabaseAsync(UriFactory.CreateDatabaseUri("FamilyDB_oa"));
```

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_oa
Press any key to continue ...
Created FamilyCollection_oa
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](#).
- A [DocumentDB account](#).
- The [GetStarted](#) 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](#).

Next steps

- Want a more complex ASP.NET MVC NoSQL tutorial? See [Build a web application with ASP.NET MVC using DocumentDB](#).
- Want to perform scale and performance testing with DocumentDB? See [Performance and Scale Testing with Azure DocumentDB](#)
- Learn how to [monitor a DocumentDB account](#).
- Run queries against our sample dataset in the [Query Playground](#).
- Learn more about the programming model in the Develop section of the [DocumentDB documentation page](#).

NoSQL tutorial: Build a DocumentDB C# console application on .NET Core

11/22/2016 • 15 min to read • [Edit on GitHub](#)

Contributors

[arramac](#) • [mimig](#)

Welcome to the NoSQL tutorial for the Azure DocumentDB .NET Core 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](#). Jump to the [Get the complete solution section](#) 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:

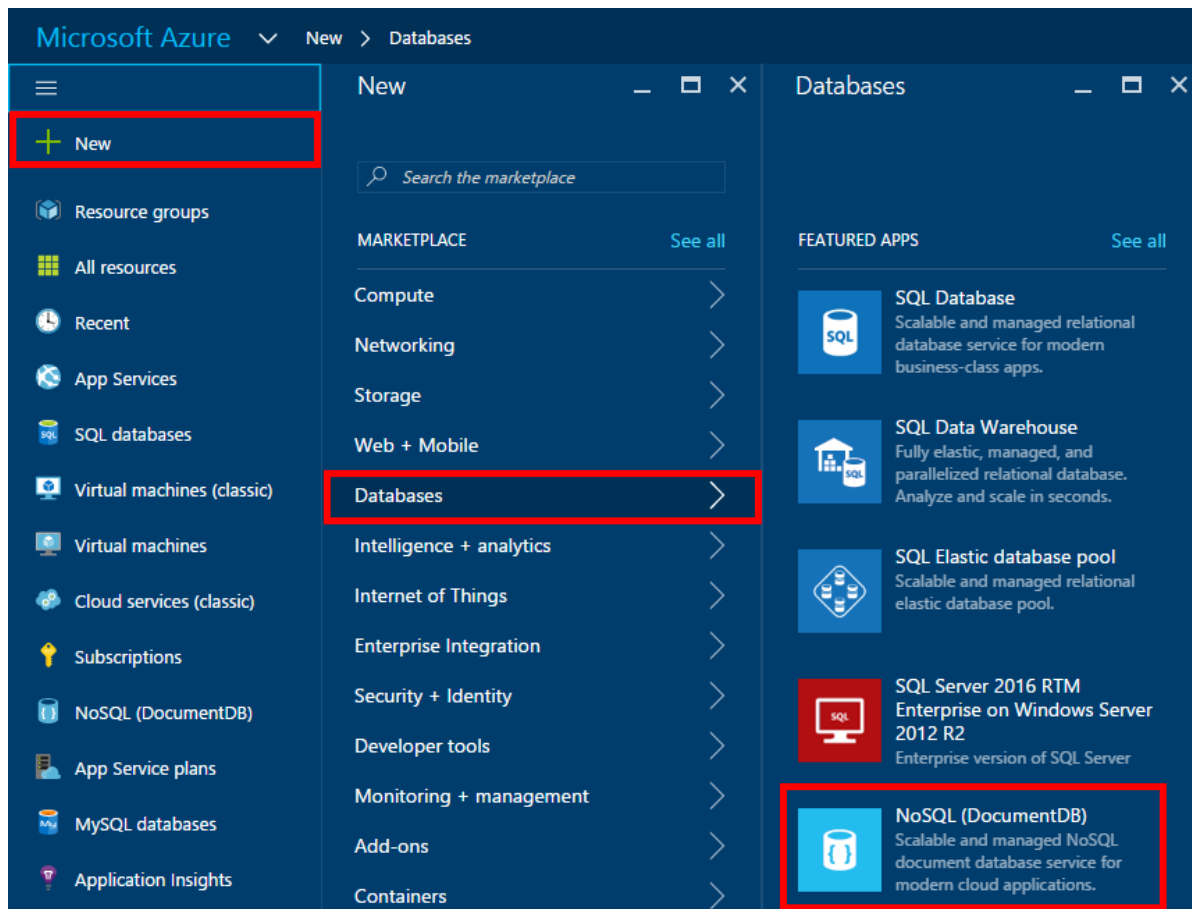
- An active Azure account. If you don't have one, you can sign up for a [free account](#).
 - Alternatively, you can use the [Azure DocumentDB Emulator](#) for this tutorial.
- [Visual Studio 2015 Update 3](#) and [.NET Core 1.0.1 - VS 2015 Tooling Preview 2](#)
 - If you're working on MacOS or Linux, you can develop .NET Core apps from the command-line by installing the [.NET Core SDK](#) for the platform of your choice.
 - If you're working on Windows, you can develop .NET Core apps from the command-line by installing the [.NET Core SDK](#).
 - You can use your own editor, or download [Visual Studio Code](#) which is free and works on Windows, Linux, and MacOS.

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](#). If you are using the DocumentDB Emulator, please follow the steps at [Azure](#)

[DocumentDB Emulator](#) to setup the emulator and skip ahead to [Setup your Visual Studio Solution](#).

1. Sign in to the [Azure portal](#).
2. In the Jumpbar, click **New**, click **Databases**, and then click **NoSQL (DocumentDB)**.



3. In the **New account** blade, specify the desired configuration for the DocumentDB account.

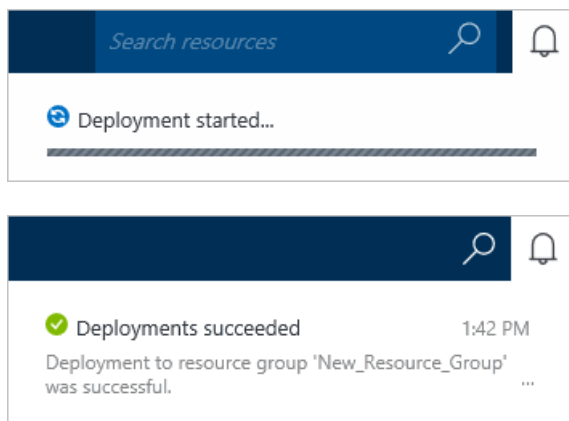
A screenshot of the 'New account' blade for NoSQL (DocumentDB) in the Azure portal. The blade title is 'NoSQL (DocumentDB)' with a subtitle 'New account'. The configuration fields are as follows:

- ID**: A text box containing 'contosoacct' with a green checkmark icon to its right. Below the text box is the URL 'documents.azure.com'.
- NoSQL API**: Two buttons, 'DocumentDB' (selected) and 'MongoDB'.
- Subscription**: A dropdown menu showing 'Visual Studio Ultimate with MSDN'.
- Resource Group**: Radio buttons for 'Create new' (selected) and 'Use existing'. Below the radio buttons is a text box containing 'contosoacct' with a green checkmark icon to its right.
- Location**: A dropdown menu showing 'West US'.
- Pin to dashboard**: A checkbox that is currently unchecked.
- Create**: A blue button.
- Automation options**: A link.

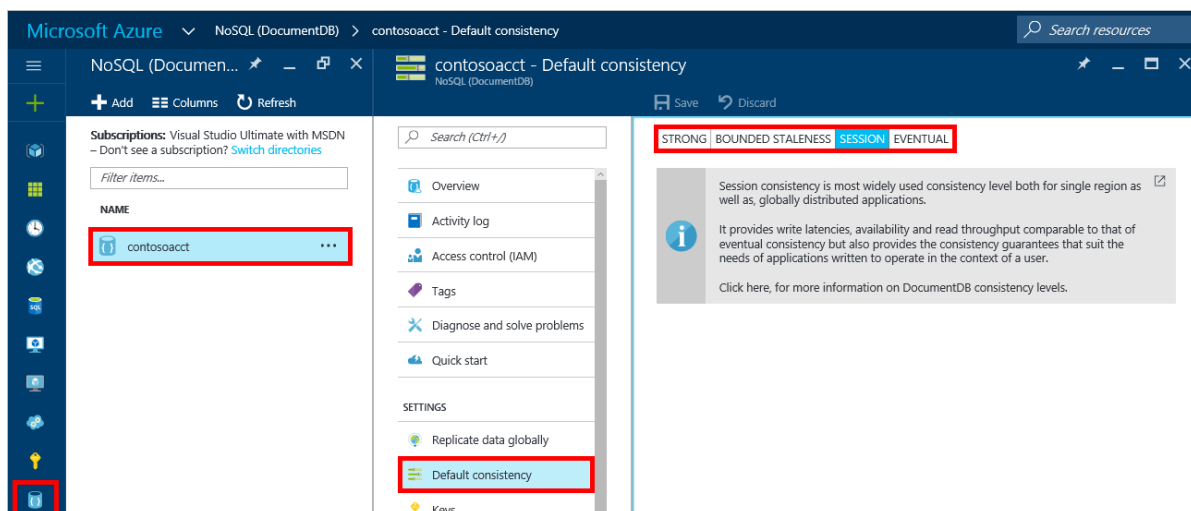
- In the ID box, enter a name to identify the DocumentDB account. When the ID is validated, a green check

mark appears in the **ID** box. The **ID** value becomes the host name within the URI. The **ID** may contain only lowercase letters, numbers, and the '-' character, and must be between 3 and 50 characters. Note that *documents.azure.com* is appended to the endpoint name you choose, the result of which becomes your DocumentDB account endpoint.

- In the **NoSQL API** box, select **DocumentDB**.
 - For **Subscription**, select the Azure subscription that you want to use for the DocumentDB account. If your account has only one subscription, that account is selected by default.
 - In **Resource Group**, select or create a resource group for your DocumentDB account. By default, a new resource group is created. For more information, see [Using the Azure portal to manage your Azure resources](#).
 - Use **Location** to specify the geographic location in which to host your DocumentDB account.
4. Once the new DocumentDB account options are configured, click **Create**. To check the status of the deployment, check the Notifications hub.



5. After the DocumentDB account is created, it is ready for use with the default settings. To review the default settings, click the **NoSQL (DocumentDB)** icon on the Jumpbar, click your new account, and then click **Default Consistency** in the Resource Menu.

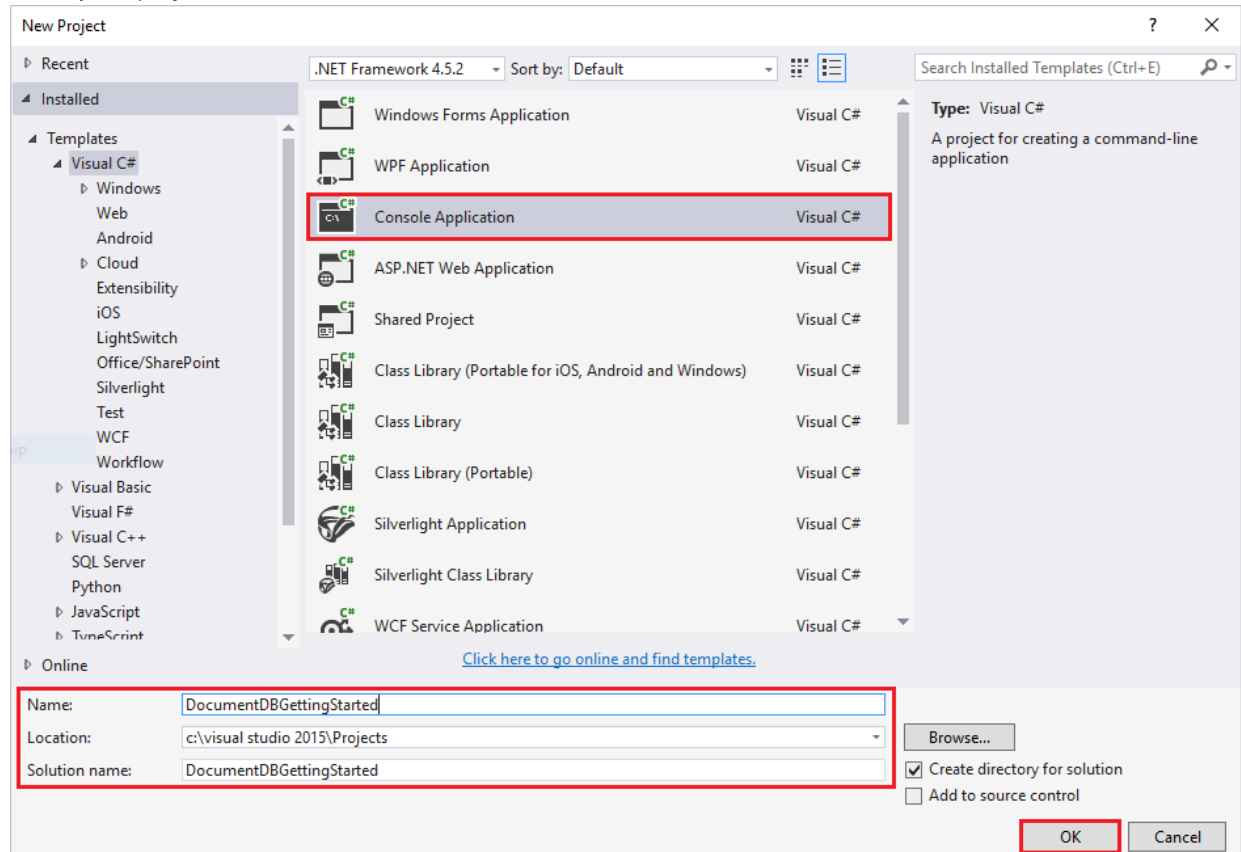


The default consistency of the DocumentDB account is set to **Session**. You can adjust the default consistency by clicking **Default Consistency** in the resource menu. To learn more about the consistency levels offered by DocumentDB, see [Consistency levels in DocumentDB](#).

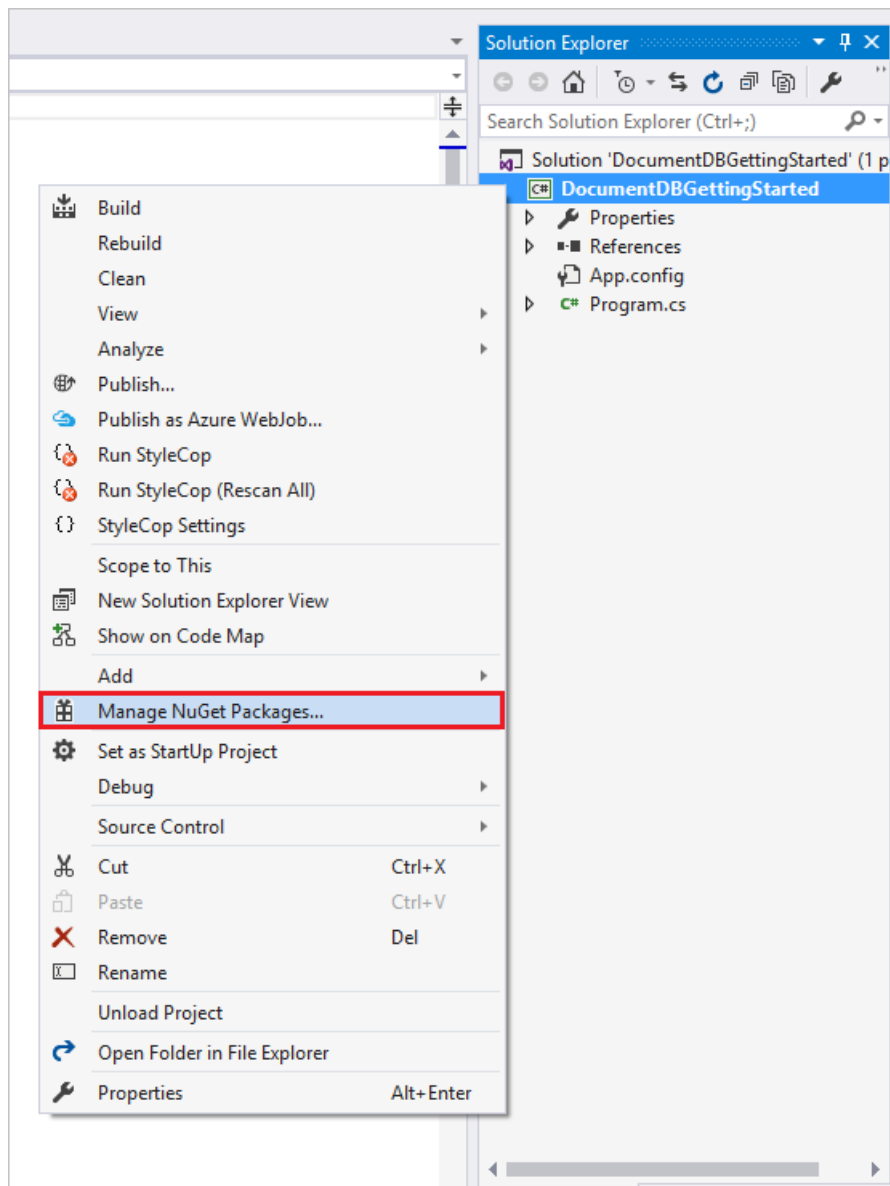
Step 2: Setup your Visual Studio solution

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# / .NET Core/Console Application (.NET Core)**,

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.Core** and click **Install**. The package ID for the DocumentDB Client Library is [Microsoft.Azure.DocumentDB.Core](#)

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](#).

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;
```

IMPORTANT

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

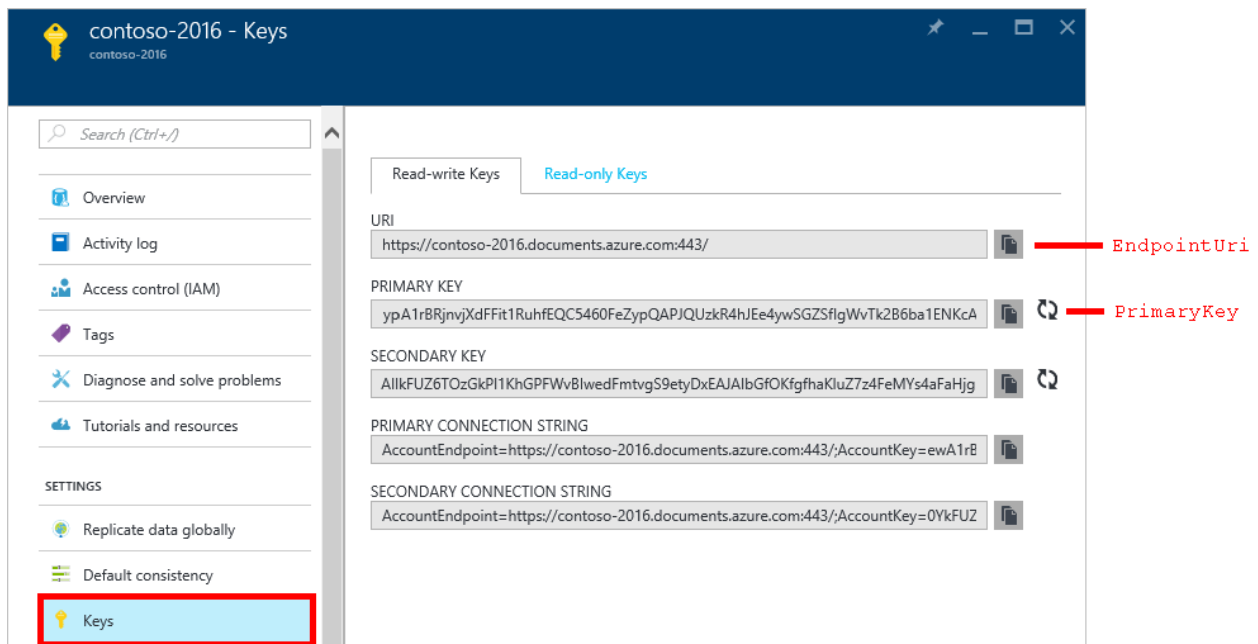
Now, add these two constants and your *client* variable underneath your public class *Program*.

```
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](#) 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, and then click **Keys**.

Copy the URI from the portal and paste it into `<your endpoint URI>` in the program.cs file. Then copy the PRIMARY KEY from the portal and paste it into `<your key>`. If you are using the Azure DocumentDB Emulator, use `https://localhost:443` as the endpoint, and the well-defined authorization key from [How to develop using the DocumentDB Emulator](#).



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](#) can be created by using the [CreateDatabaseAsync](#) 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_oa");
}
```

Press **F5** to run your application.

Congratulations! You have successfully created a DocumentDB database.

Step 5: Create a collection

WARNING

CreateDocumentCollectionAsync will create a new collection with reserved throughput, which has pricing implications. For more details, please visit our [pricing page](#).

A **collection** can be created by using the **CreateDocumentCollectionAsync** 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),
                collectionInfo,
                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_oa*.

```
this.client = new DocumentClient(new Uri(EndpointUri), PrimaryKey);

await this.CreateDatabaseIfNotExists("FamilyDB_oa");

// ADD THIS PART TO YOUR CODE
await this.CreateDocumentCollectionIfNotExists("FamilyDB_oa", "FamilyCollection_oa");
```

Press **F5** to run your application.

Congratulations! You have successfully created a DocumentDB document collection.

Step 6: Create JSON documents

A [document](#) can be created by using the [CreateDocumentAsync](#) 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](#).

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_oa");

await this.CreateDocumentCollectionIfNotExists("FamilyDB_oa", "FamilyCollection_oa");

// 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_oa", "FamilyCollection_oa", andersenFamily);

Family wakefieldFamily = new Family
{
    Id = "Wakefield.7",
    LastName = "Wakefield",
    Parents = new Parent[]
```

```

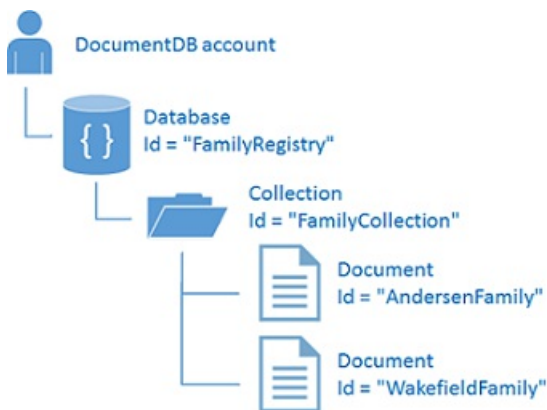
Parents = new Parent[]
{
    new Parent { FamilyName = "Wakefield", FirstName = "Robin" },
    new Parent { FamilyName = "Miller", FirstName = "Ben" }
},
Children = new Child[]
{
    new Child
    {
        FamilyName = "Merriam",
        FirstName = "Jesse",
        Gender = "female",
        Grade = 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_oa", "FamilyCollection_oa", wakefieldFamily);

```

Press **F5** to run your application.

Congratulations! You have successfully created two DocumentDB documents.



Step 7: Query DocumentDB resources

DocumentDB supports rich [queries](#) 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.

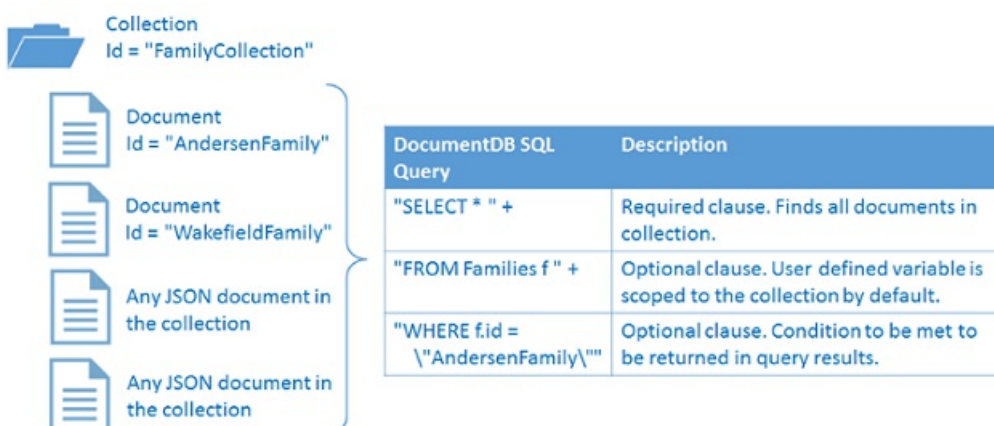
```
await this.CreateFamilyDocumentIfNotExists("FamilyDB_oa", "FamilyCollection_oa", wakefieldFamily);

// ADD THIS PART TO YOUR CODE
this.ExecuteSimpleQuery("FamilyDB_oa", "FamilyCollection_oa");
```

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.



The **FROM** 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;
    }
}
```

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_oa", "FamilyCollection_oa", wakefieldFamily);

this.ExecuteSimpleQuery("FamilyDB_oa", "FamilyCollection_oa");

// ADD THIS PART TO YOUR CODE
// Update the Grade of the Andersen Family child
andersenFamily.Children[0].Grade = 6;

await this.ReplaceFamilyDocument("FamilyDB_oa", "FamilyCollection_oa", "Andersen.1", andersenFamily);

this.ExecuteSimpleQuery("FamilyDB_oa", "FamilyCollection_oa");
```

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;
    }
}
```

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

```
await this.ReplaceFamilyDocument("FamilyDB_oa", "FamilyCollection_oa", "Andersen.1", andersenFamily);

this.ExecuteSimpleQuery("FamilyDB_oa", "FamilyCollection_oa");

// ADD THIS PART TO CODE
await this.DeleteFamilyDocument("FamilyDB_oa", "FamilyCollection_oa", "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_oa", "FamilyCollection_oa");

await this.DeleteFamilyDocument("FamilyDB_oa", "FamilyCollection_oa", "Andersen.1");

// ADD THIS PART TO CODE
// Clean up/delete the database
await this.client.DeleteDatabaseAsync(UriFactory.CreateDatabaseUri("FamilyDB_oa"));
```

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_oa
Press any key to continue ...
Created FamilyCollection_oa
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](#).
- A [DocumentDB account](#).
- The [GetStarted](#) solution available on GitHub.

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

Next steps

- Want a more complex ASP.NET MVC NoSQL tutorial? See [Build a web application with ASP.NET MVC using DocumentDB](#).
- Want to perform scale and performance testing with DocumentDB? See [Performance and Scale Testing with Azure DocumentDB](#)
- Learn how to [monitor a DocumentDB account](#).
- Run queries against our sample dataset in the [Query Playground](#).
- Learn more about the programming model in the Develop section of the [DocumentDB documentation page](#).

NoSQL Node.js tutorial: DocumentDB Node.js console application

11/22/2016 • 13 min to read • [Edit on GitHub](#)

Contributors

[Andrew Hoh](#) • [mimig](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [arramac](#) • [Guy Burstein](#) • [v-aljenk](#)

Welcome to the Node.js tutorial for the Azure DocumentDB Node.js SDK! After following this tutorial, you'll have a console application that creates and queries DocumentDB resources, including a Node database.

We'll cover:

- Creating and connecting to a DocumentDB account
- Setting up your application
- Creating a node database
- Creating a collection
- Creating JSON documents
- Querying the collection
- Replacing a document
- Deleting a document
- Deleting the node database

Don't have time? Don't worry! The complete solution is available on [GitHub](#). See [Get the complete solution](#) for quick instructions.

After you've completed the Node.js tutorial, please use the voting buttons at the top and 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 for the Node.js tutorial

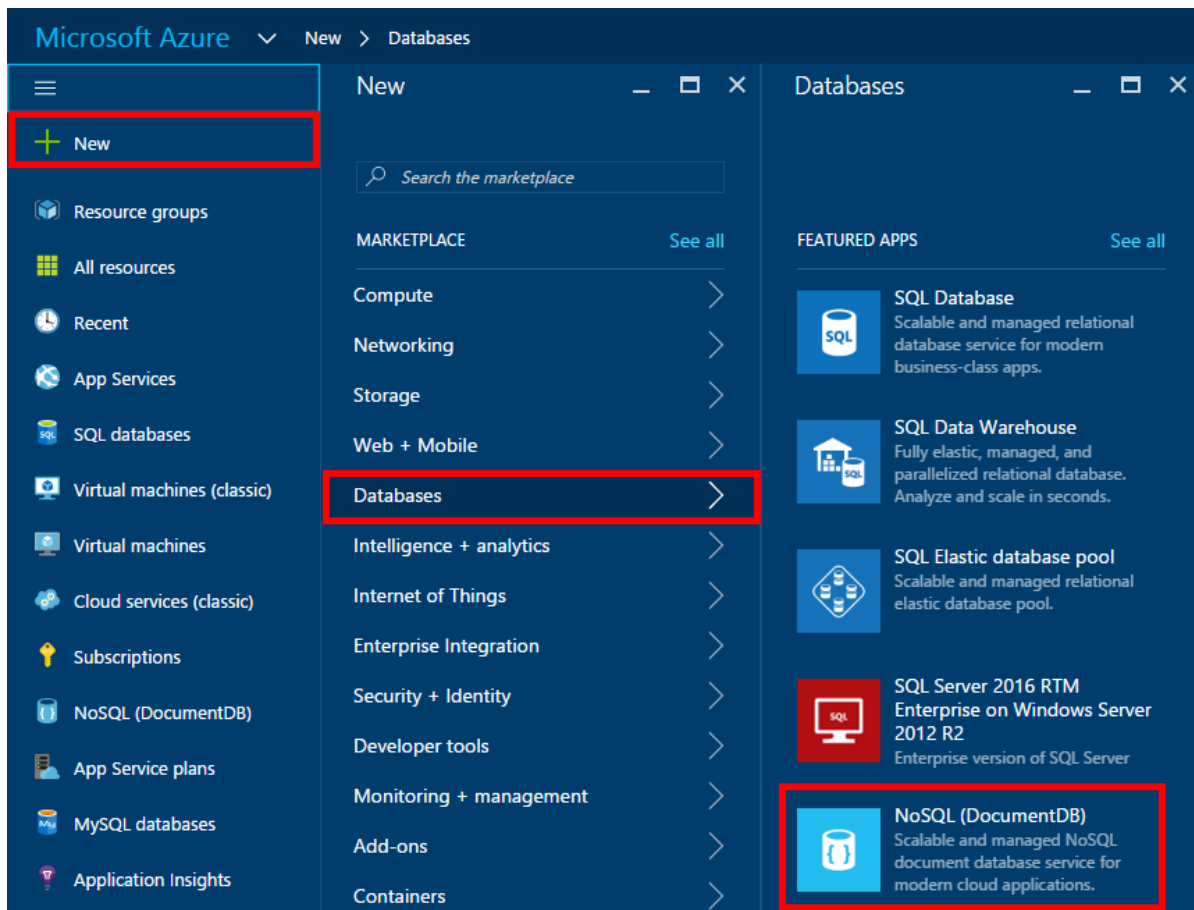
Please make sure you have the following:

- An active Azure account. If you don't have one, you can sign up for a [Free Azure Trial](#).
 - Alternatively, you can use the [Azure DocumentDB Emulator](#) for this tutorial.
- [Node.js](#) version v0.10.29 or higher.

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 Node.js application](#). If you are using the DocumentDB Emulator, please follow the steps at [Azure DocumentDB Emulator](#) to setup the emulator and skip ahead to [Setup your Node.js application](#).

1. Sign in to the [Azure portal](#).
2. In the Jumpbar, click **New**, click **Databases**, and then click **NoSQL (DocumentDB)**.



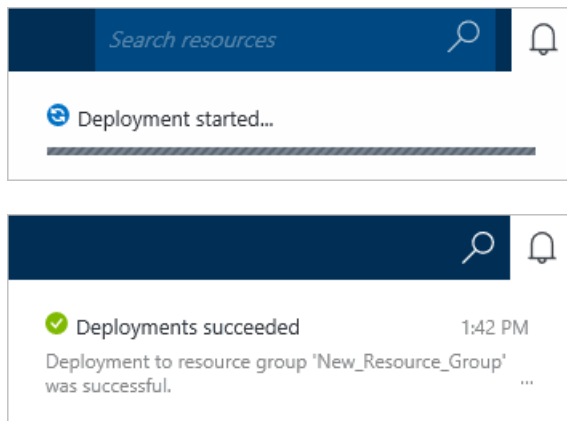
3. In the **New account** blade, specify the desired configuration for the DocumentDB account.

The screenshot shows the 'NoSQL (DocumentDB) New account' blade. The configuration fields are as follows:

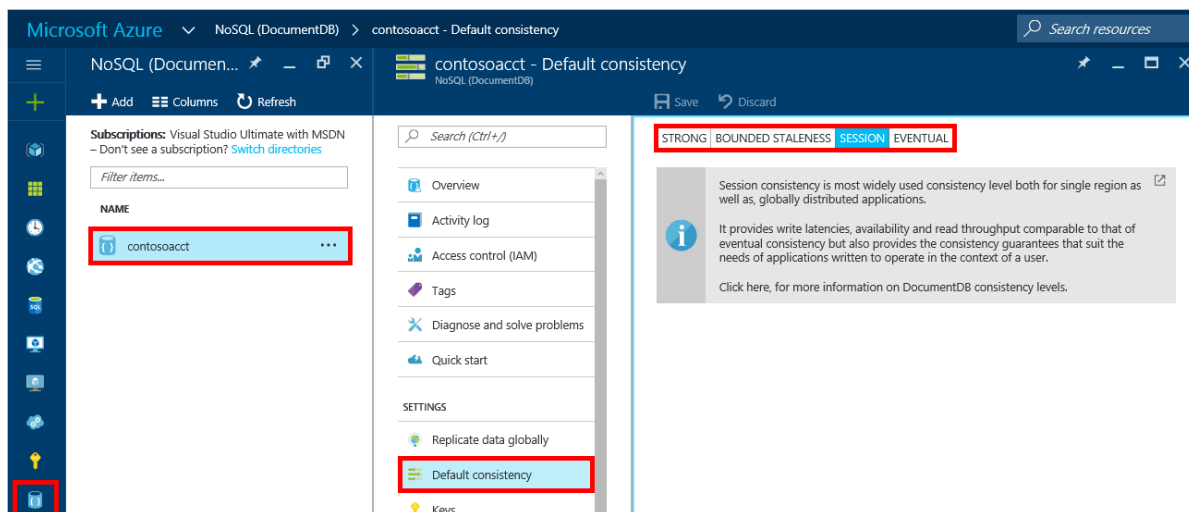
- ID:** A text box containing 'contosoacct' with a green checkmark indicating validation. Below it, the URL 'documents.azure.com' is displayed.
- NoSQL API:** Two tabs: 'DocumentDB' (selected) and 'MongoDB'.
- Subscription:** A dropdown menu showing 'Visual Studio Ultimate with MSDN'.
- Resource Group:** Radio buttons for 'Create new' (selected) and 'Use existing'. Below, a text box contains 'contosoacct' with a green checkmark.
- Location:** A dropdown menu showing 'West US'.
- Pin to dashboard:** An unchecked checkbox.
- Buttons:** A blue 'Create' button and a blue link for 'Automation options'.

- In the ID box, enter a name to identify the DocumentDB account. When the ID is validated, a green check mark appears in the ID box. The ID value becomes the host name within the URI. The ID may contain only lowercase letters, numbers, and the '-' character, and must be between 3 and 50 characters. Note that *documents.azure.com* is appended to the endpoint name you choose, the result of which becomes your DocumentDB account endpoint.

- In the **NoSQL API** box, select **DocumentDB**.
 - For **Subscription**, select the Azure subscription that you want to use for the DocumentDB account. If your account has only one subscription, that account is selected by default.
 - In **Resource Group**, select or create a resource group for your DocumentDB account. By default, a new resource group is created. For more information, see [Using the Azure portal to manage your Azure resources](#).
 - Use **Location** to specify the geographic location in which to host your DocumentDB account.
4. Once the new DocumentDB account options are configured, click **Create**. To check the status of the deployment, check the Notifications hub.



5. After the DocumentDB account is created, it is ready for use with the default settings. To review the default settings, click the **NoSQL (DocumentDB)** icon on the Jumpbar, click your new account, and then click **Default Consistency** in the Resource Menu.



The default consistency of the DocumentDB account is set to **Session**. You can adjust the default consistency by clicking **Default Consistency** in the resource menu. To learn more about the consistency levels offered by DocumentDB, see [Consistency levels in DocumentDB](#).

Step 2: Setup your Node.js application

1. Open your favorite terminal.
2. Locate the folder or directory where you'd like to save your Node.js application.
3. Create two empty JavaScript files with the following commands:
 - Windows:
 - `fsutil file createnew app.js 0`
 - `fsutil file createnew config.js 0`
 - Linux/OS X:

- `touch app.js`
- `touch config.js`

4. Install the `documentdb` module via npm. Use the following command:

- `npm install documentdb --save`

Great! Now that you've finished setting up, let's start writing some code.

Step 3: Set your app's configurations

Open `config.js` in your favorite text editor.

Then, copy and paste the code snippet below and set properties `config.endpoint` and `config.primaryKey` to your DocumentDB endpoint uri and primary key. Both these configurations can be found in the [Azure Portal](#).

The screenshot shows the Azure Portal interface for a DocumentDB account. The left sidebar contains a 'Keys' tab, which is highlighted with a red rectangle. The main content area displays the following fields:

- URI:** `https://contoso1.documents.azure.com:4` (labeled `config.endpoint`)
- PRIMARY KEY:** `8B2073E4-B885-4360-80A0-108AA34F5A52` (labeled `config.primaryKey`)
- SECONDARY KEY:** `8B2073E4-B885-4360-80A0-108AA34F5A52`
- PRIMARY CONNECTION STRING:** `AccountEndpoint=https://contoso1.docu`
- SECONDARY CONNECTION STRING:** `AccountEndpoint=https://contoso1.docu`
- READ-ONLY KEYS:** Manage read-only keys

```
// ADD THIS PART TO YOUR CODE
var config = {}

config.endpoint = "~your DocumentDB endpoint uri here~";
config.primaryKey = "~your primary key here~";
```

Copy and paste the `database id`, `collection id`, and `JSON documents` to your `config` object below where you set your `config.endpoint` and `config.authKey` properties. If you already have data you'd like to store in your database, you can use DocumentDB's [Data Migration tool](#) rather than adding the document definitions.

```
config.endpoint = "~your DocumentDB endpoint uri here~";
config.primaryKey = "~your primary key here~";
```

```
// ADD THIS PART TO YOUR CODE
config.database = {
  "id": "FamilyDB"
};

config.collection = {
  "id": "FamilyColl"
};

config.documents = {
  "Andersen": {
    "id": "Anderson.1",
    "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"
    }
  },
  "Wakefield": {
    "id": "Wakefield.7",
    "parents": [{
      "familyName": "Wakefield",
      "firstName": "Robin"
    }, {
      "familyName": "Miller",
      "firstName": "Ben"
    }],
    "children": [{
      "familyName": "Merriam",
      "firstName": "Jesse",
      "gender": "female",
      "grade": 8,
      "pets": [{
        "givenName": "Goofy"
      }], {
        "givenName": "Shadow"
      }]
    }, {
      "familyName": "Miller",
      "firstName": "Lisa",
      "gender": "female",
      "grade": 1
    }],
    "address": {
      "state": "NY",
      "county": "Manhattan",
      "city": "NY"
    },
    "isRegistered": false
  }
};
```

The database, collection, and document definitions will act as your DocumentDB `database id`, `collection id`, and

documents' data.

Finally, export your `config` object, so that you can reference it within the `app.js` file.

```
    },
    "isRegistered": false
  }
};

// ADD THIS PART TO YOUR CODE
module.exports = config;
```

Step 4: Connect to a DocumentDB account

Open your empty `app.js` file in the text editor. Copy and paste the code below to import the `documentdb` module and your newly created `config` module.

```
// ADD THIS PART TO YOUR CODE
"use strict";

var documentClient = require("documentdb").DocumentClient;
var config = require("./config");
var url = require('url');
```

Copy and paste the code to use the previously saved `config.endpoint` and `config.primaryKey` to create a new `DocumentClient`.

```
var config = require("./config");
var url = require('url');

// ADD THIS PART TO YOUR CODE
var client = new documentClient(config.endpoint, { "masterKey": config.primaryKey });
```

Now that you have the code to initialize the `documentdb` client, let's take a look at working with DocumentDB resources.

Step 5: Create a Node database

Copy and paste the code below to set the HTTP status for Not Found, the database url, and the collection url. These urls are how the DocumentDB client will find the right database and collection.

```
var client = new documentClient(config.endpoint, { "masterKey": config.primaryKey });

// ADD THIS PART TO YOUR CODE
var HttpStatusCodes = { NOTFOUND: 404 };
var databaseUrl = `dbs/${config.database.id}`;
var collectionUrl = `${databaseUrl}/colls/${config.collection.id}`;
```

A [database](#) can be created by using the [createDatabase](#) function of the `DocumentClient` class. A database is the logical container of document storage partitioned across collections.

Copy and paste the `getDatabase` function for creating your new database in the `app.js` file with the `id` specified in the `config` object. The function will check if the database with the same `FamilyRegistry` id does not already exist. If it does exist, we'll return that database instead of creating a new one.

```

var collectionUrl = `${databaseUrl}/colls/${config.collection.id}`;

// ADD THIS PART TO YOUR CODE
function getDatabase() {
  console.log(`Getting database:\n${config.database.id}\n`);

  return new Promise((resolve, reject) => {
    client.readDatabase(databaseUrl, (err, result) => {
      if (err) {
        if (err.code == HttpStatusCodes.NOTFOUND) {
          client.createDatabase(config.database, (err, created) => {
            if (err) reject(err)
            else resolve(created);
          });
        } else {
          reject(err);
        }
      } else {
        resolve(result);
      }
    });
  });
}

```

Copy and paste the code below where you set the **getDatabase** function to add the helper function **exit** that will print the exit message and the call to **getDatabase** function.

```

      } else {
        resolve(result);
      }
    });
  });
}

// ADD THIS PART TO YOUR CODE
function exit(message) {
  console.log(message);
  console.log('Press any key to exit');
  process.stdin.setRawMode(true);
  process.stdin.resume();
  process.stdin.on('data', process.exit.bind(process, 0));
}

getDatabase()
  .then(() => { exit(`Completed successfully`); })
  .catch((error) => { exit(`Completed with error ${JSON.stringify(error)}`); });

```

In your terminal, locate your `app.js` file and run the command: `node app.js`

Congratulations! You have successfully created a DocumentDB database.

Step 6: Create a collection

WARNING

`CreateDocumentCollectionAsync` will create a new collection, which has pricing implications. For more details, please visit [our pricing page](#).

A [collection](#) can be created by using the `createCollection` function of the `DocumentClient` class. A collection is a container of JSON documents and associated JavaScript application logic.

Copy and paste the **getCollection** function underneath the **getDatabase** function for creating your new

collection with the `id` specified in the `config` object. Again, we'll check to make sure a collection with the same `FamilyCollection` id does not already exist. If it does exist, we'll return that collection instead of creating a new one.

```
        } else {
            resolve(result);
        }
    });
});
}

// ADD THIS PART TO YOUR CODE
function getCollection() {
    console.log(`Getting collection:\n${config.collection.id}\n`);

    return new Promise((resolve, reject) => {
        client.readCollection(collectionUrl, (err, result) => {
            if (err) {
                if (err.code == HttpStatusCodes.NOTFOUND) {
                    client.createCollection(databaseUrl, config.collection, { offerThroughput: 400 }, (err,
created) => {
                        if (err) reject(err)
                        else resolve(created);
                    });
                } else {
                    reject(err);
                }
            } else {
                resolve(result);
            }
        });
    });
}
```

Copy and paste the code below the call to **getDatabase** to execute the **getCollection** function.

```
getDatabase()

// ADD THIS PART TO YOUR CODE
.then(() => getCollection())
// ENDS HERE

.then(() => { exit(`Completed successfully`); })
.catch((error) => { exit(`Completed with error ${JSON.stringify(error)}`) });
```

In your terminal, locate your `app.js` file and run the command: `node app.js`

Congratulations! You have successfully created a DocumentDB collection.

Step 7: Create a document

A [document](#) can be created by using the [createDocument](#) function of the **DocumentClient** class. Documents are user defined (arbitrary) JSON content. You can now insert a document into DocumentDB.

Copy and paste the **getFamilyDocument** function underneath the **getCollection** function for creating the documents containing the JSON data saved in the `config` object. Again, we'll check to make sure a document with the same id does not already exist.

```

        } else {
            resolve(result);
        }
    });
});
}

// ADD THIS PART TO YOUR CODE
function getFamilyDocument(document) {
    let documentUrl = `${collectionUrl}/docs/${document.id}`;
    console.log(`Getting document:\n${document.id}\n`);

    return new Promise((resolve, reject) => {
        client.readDocument(documentUrl, { partitionKey: document.district }, (err, result) => {
            if (err) {
                if (err.code == HttpStatusCodes.NOTFOUND) {
                    client.createDocument(collectionUrl, document, (err, created) => {
                        if (err) reject(err)
                        else resolve(created);
                    });
                } else {
                    reject(err);
                }
            } else {
                resolve(result);
            }
        });
    });
};
};
};

```

Copy and paste the code below the call to **getCollection** to execute the **getFamilyDocument** function.

```

getDatabase()
.then(() => getCollection())

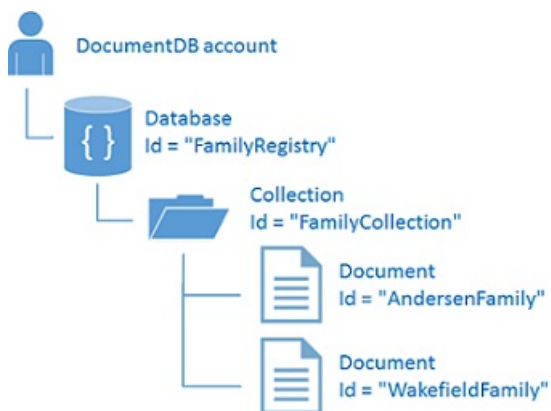
// ADD THIS PART TO YOUR CODE
.then(() => getFamilyDocument(config.documents.Andersen))
.then(() => getFamilyDocument(config.documents.Wakefield))
// ENDS HERE

.then(() => { exit(`Completed successfully`); })
.catch((error) => { exit(`Completed with error ${JSON.stringify(error)}`) });

```

In your terminal, locate your `app.js` file and run the command: `node app.js`

Congratulations! You have successfully created a DocumentDB documents.



Step 8: Query DocumentDB resources

DocumentDB supports [rich queries](#) against JSON documents stored in each collection. The following sample code

shows a query that you can run against the documents in your collection.

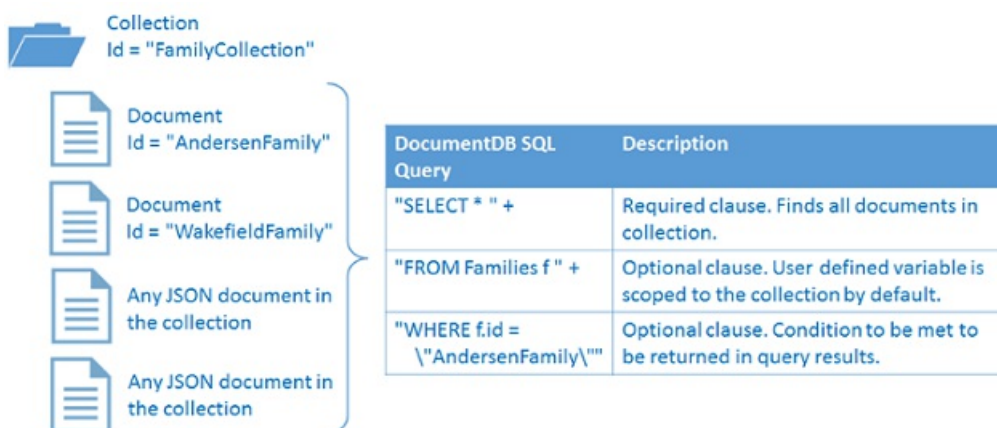
Copy and paste the **queryCollection** function underneath the **getFamilyDocument** function. DocumentDB supports SQL-like queries as shown below. For more information on building complex queries, check out the [Query Playground](#) and the [query documentation](#).

```
        } else {
            resolve(result);
        }
    });
});
}

// ADD THIS PART TO YOUR CODE
function queryCollection() {
    console.log(`Querying collection through index:\n${config.collection.id}`);

    return new Promise((resolve, reject) => {
        client.queryDocuments(
            collectionUrl,
            'SELECT VALUE r.children FROM root r WHERE r.lastName = "Andersen"'
        ).toArray((err, results) => {
            if (err) reject(err)
            else {
                for (var queryResult of results) {
                    let resultString = JSON.stringify(queryResult);
                    console.log(`\tQuery returned ${resultString}`);
                }
                console.log();
                resolve(results);
            }
        });
    });
};
```

The following diagram illustrates how the DocumentDB SQL query syntax is called against the collection you created.



The **FROM** 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.

Copy and paste the code below the call to **getFamilyDocument** to execute the **queryCollection** function.

```

.then(() => getFamilyDocument(config.documents.Andersen))
.then(() => getFamilyDocument(config.documents.Wakefield))

// ADD THIS PART TO YOUR CODE
.then(() => queryCollection())
// ENDS HERE

.then(() => { exit(`Completed successfully`); })
.catch((error) => { exit(`Completed with error ${JSON.stringify(error)}`) });

```

In your terminal, locate your `app.js` file and run the command: `node app.js`

Congratulations! You have successfully queried DocumentDB documents.

Step 9: Replace a document

DocumentDB supports replacing JSON documents.

Copy and paste the **replaceDocument** function underneath the **queryCollection** function.

```

        }
        console.log();
        resolve(result);
    }
    });
});
}

// ADD THIS PART TO YOUR CODE
function replaceFamilyDocument(document) {
    let documentUrl = `${collectionUrl}/docs/${document.id}`;
    console.log(`Replacing document:\n${document.id}\n`);
    document.children[0].grade = 6;

    return new Promise((resolve, reject) => {
        client.replaceDocument(documentUrl, document, (err, result) => {
            if (err) reject(err);
            else {
                resolve(result);
            }
        });
    });
};
};
};

```

Copy and paste the code below the call to **queryCollection** to execute the **replaceDocument** function. Also, add the code to call **queryCollection** again to verify that the document had successfully changed.

```

.then(() => getFamilyDocument(config.documents.Andersen))
.then(() => getFamilyDocument(config.documents.Wakefield))
.then(() => queryCollection())

// ADD THIS PART TO YOUR CODE
.then(() => replaceFamilyDocument(config.documents.Andersen))
.then(() => queryCollection())
// ENDS HERE

.then(() => { exit(`Completed successfully`); })
.catch((error) => { exit(`Completed with error ${JSON.stringify(error)}`) });

```

In your terminal, locate your `app.js` file and run the command: `node app.js`

Congratulations! You have successfully replaced a DocumentDB document.

Step 10: Delete a document

DocumentDB supports deleting JSON documents.

Copy and paste the **deleteDocument** function underneath the **replaceDocument** function.

```
        else {
            resolve(result);
        }
    });
});
};

// ADD THIS PART TO YOUR CODE
function deleteFamilyDocument(document) {
    let documentUrl = `${collectionUrl}/docs/${document.id}`;
    console.log(`Deleting document:\n${document.id}\n`);

    return new Promise((resolve, reject) => {
        client.deleteDocument(documentUrl, (err, result) => {
            if (err) reject(err);
            else {
                resolve(result);
            }
        });
    });
};
};
```

Copy and paste the code below the call to the second **queryCollection** to execute the **deleteDocument** function.

```
.then(() => queryCollection())
.then(() => replaceFamilyDocument(config.documents.Andersen))
.then(() => queryCollection())

// ADD THIS PART TO YOUR CODE
.then(() => deleteFamilyDocument(config.documents.Andersen))
// ENDS HERE

.then(() => { exit(`Completed successfully`); })
.catch((error) => { exit(`Completed with error ${JSON.stringify(error)}`) });
```

In your terminal, locate your `app.js` file and run the command: `node app.js`

Congratulations! You have successfully deleted a DocumentDB document.

Step 11: Delete the Node database

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

Copy and paste the following code snippet (function **cleanup**) to remove the database and all the children resources.

```

        else {
            resolve(result);
        }
    });
});
};

// ADD THIS PART TO YOUR CODE
function cleanup() {
    console.log(`Cleaning up by deleting database ${config.database.id}`);

    return new Promise((resolve, reject) => {
        client.deleteDatabase(databaseUrl, (err) => {
            if (err) reject(err)
            else resolve(null);
        });
    });
}
}

```

Copy and paste the code below the call to **deleteDocument** to execute the **cleanup** function.

```

.then(() => deleteFamilyDocument(config.documents.Andersen))

// ADD THIS PART TO YOUR CODE
.then(() => cleanup())
// ENDS HERE

.then(() => { exit(`Completed successfully`); })
.catch((error) => { exit(`Completed with error ${JSON.stringify(error)}`) });

```

Step 12: Run your Node.js application all together!

Altogether, the sequence for calling your functions should look like this:

```

getDatabase()
.then(() => getCollection())
.then(() => getFamilyDocument(config.documents.Andersen))
.then(() => getFamilyDocument(config.documents.Wakefield))
.then(() => queryCollection())
.then(() => replaceFamilyDocument(config.documents.Andersen))
.then(() => queryCollection())
.then(() => deleteFamilyDocument(config.documents.Andersen))
.then(() => cleanup())
.then(() => { exit(`Completed successfully`); })
.catch((error) => { exit(`Completed with error ${JSON.stringify(error)}`) });

```

In your terminal, locate your `app.js` file and run the command: `node app.js`

You should see the output of your get started app. The output should match the example text below.

```
Getting database:
FamilyDB

Getting collection:
FamilyColl

Getting document:
Anderson.1

Getting document:
Wakefield.7

Querying collection through index:
FamilyColl
  Query returned [{"firstName":"Henriette Thaulow","gender":"female","grade":5,"pets":
[{"givenName":"Fluffy"}]}]

Replacing document:
Anderson.1

Querying collection through index:
FamilyColl
  Query returned [{"firstName":"Henriette Thaulow","gender":"female","grade":6,"pets":
[{"givenName":"Fluffy"}]}]

Deleting document:
Anderson.1

Cleaning up by deleting database FamilyDB
Completed successfully
Press any key to exit
```

Congratulations! You've created you've completed the Node.js tutorial and have your first DocumentDB console application!

Get the complete Node.js tutorial solution

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

- [DocumentDB account](#).
- The [GetStarted](#) solution available on GitHub.

Install the **documentdb** module via npm. Use the following command:

- `npm install documentdb --save`

Next, in the `config.js` file, update the `config.endpoint` and `config.authKey` values as described in [Step 3: Set your app's configurations](#).

Next steps

- Want a more complex Node.js sample? See [Build a Node.js web application using DocumentDB](#).
- Learn how to [monitor a DocumentDB account](#).
- Run queries against our sample dataset in the [Query Playground](#).
- Learn more about the programming model in the Develop section of the [DocumentDB documentation page](#).

NoSQL C++ tutorial: DocumentDB C++ console application

11/22/2016 • 8 min to read • [Edit on GitHub](#)

Contributors

[Ankit Asthana](#) • [mimig](#) • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#)

Welcome to the C++ tutorial for the Azure DocumentDB endorsed SDK for C++! After following this tutorial, you'll have a console application that creates and queries DocumentDB resources, including a C++ database.

We'll cover:

- Creating and connecting to a DocumentDB account
- Setting up your application
- Creating a C++ DocumentDB database
- Creating a collection
- Creating JSON documents
- Querying the collection
- Replacing a document
- Deleting a document
- Deleting the C++ DocumentDB database

Don't have time? Don't worry! The complete solution is available on [GitHub](#). See [Get the complete solution](#) for quick instructions.

After you've completed the C++ tutorial, please use the voting buttons at the 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 or [reach out to us here](#).

Now let's get started!

Prerequisites for the C++ tutorial

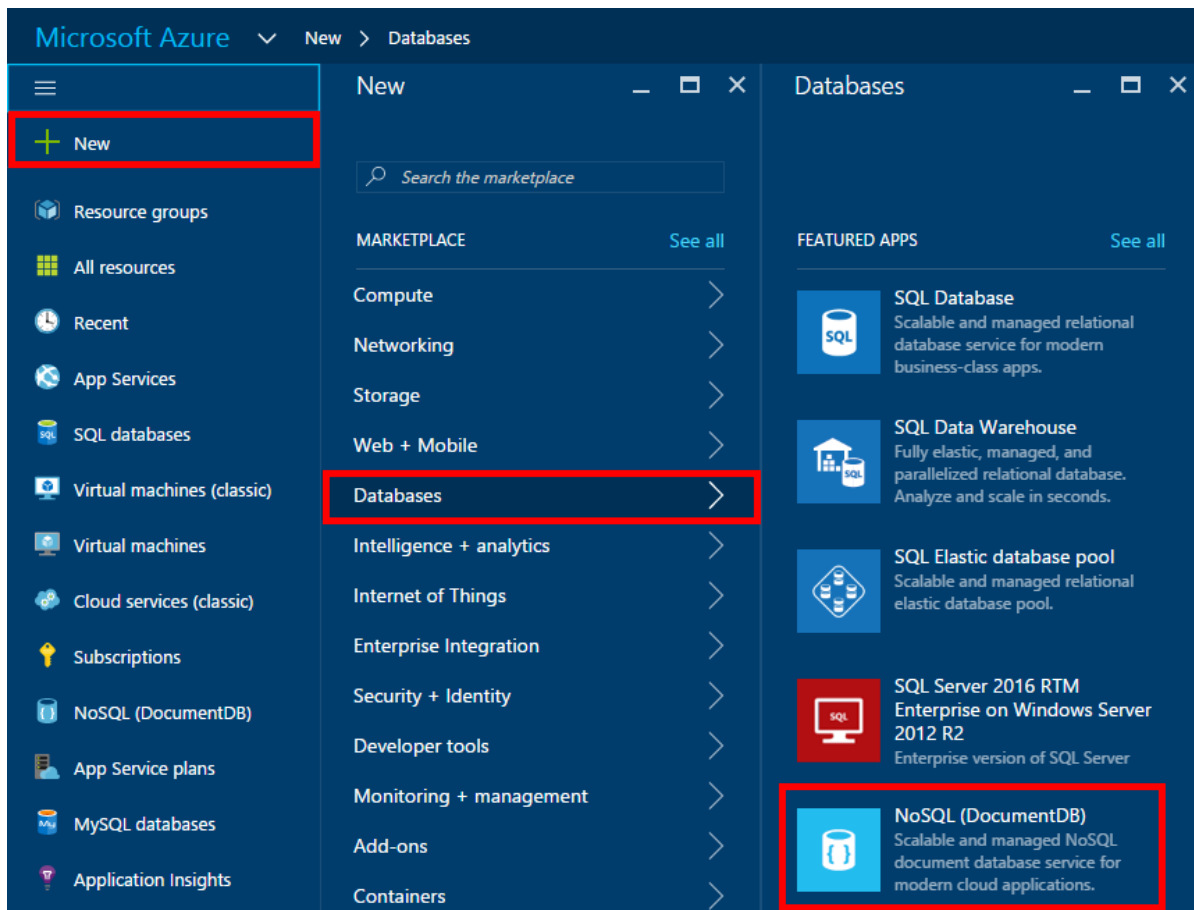
Please make sure you have the following:

- An active Azure account. If you don't have one, you can sign up for a [Free Azure Trial](#).
- [Visual Studio](#), with the C++ language components installed.

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 C++ application](#).

1. Sign in to the [Azure portal](#).
2. In the Jumpbar, click **New**, click **Databases**, and then click **NoSQL (DocumentDB)**.



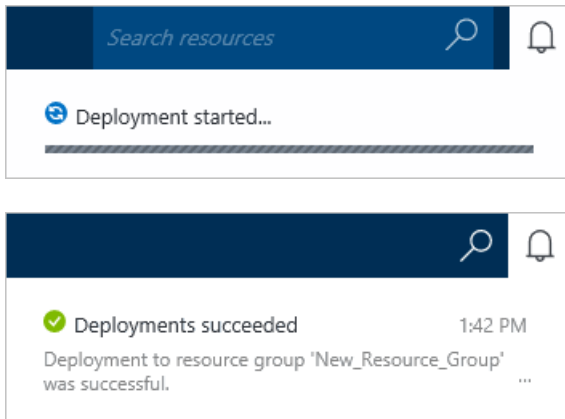
3. In the **New account** blade, specify the desired configuration for the DocumentDB account.

The screenshot shows the 'NoSQL (DocumentDB) New account' blade. The form contains the following fields and options:

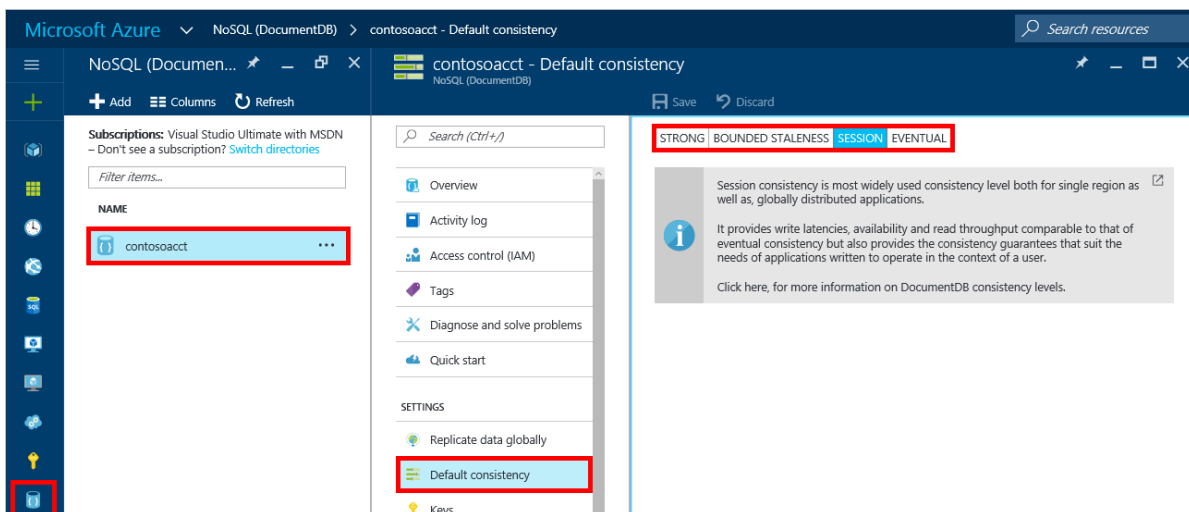
- ID:** A text box containing 'contosoacct' with a green checkmark indicating validation. Below it, the text 'documents.azure.com' is displayed.
- NoSQL API:** Two buttons: 'DocumentDB' (selected) and 'MongoDB'.
- Subscription:** A dropdown menu showing 'Visual Studio Ultimate with MSDN'.
- Resource Group:** Radio buttons for 'Create new' (selected) and 'Use existing'. Below, a text box contains 'contosoacct' with a green checkmark.
- Location:** A dropdown menu showing 'West US'.
- Pin to dashboard:** An unchecked checkbox.
- Buttons:** 'Create' and 'Automation options'.

- In the ID box, enter a name to identify the DocumentDB account. When the ID is validated, a green check mark appears in the ID box. The ID value becomes the host name within the URI. The ID may contain only lowercase letters, numbers, and the '-' character, and must be between 3 and 50 characters. Note that *documents.azure.com* is appended to the endpoint name you choose, the result of which becomes your DocumentDB account endpoint.

- In the **NoSQL API** box, select **DocumentDB**.
 - For **Subscription**, select the Azure subscription that you want to use for the DocumentDB account. If your account has only one subscription, that account is selected by default.
 - In **Resource Group**, select or create a resource group for your DocumentDB account. By default, a new resource group is created. For more information, see [Using the Azure portal to manage your Azure resources](#).
 - Use **Location** to specify the geographic location in which to host your DocumentDB account.
4. Once the new DocumentDB account options are configured, click **Create**. To check the status of the deployment, check the Notifications hub.



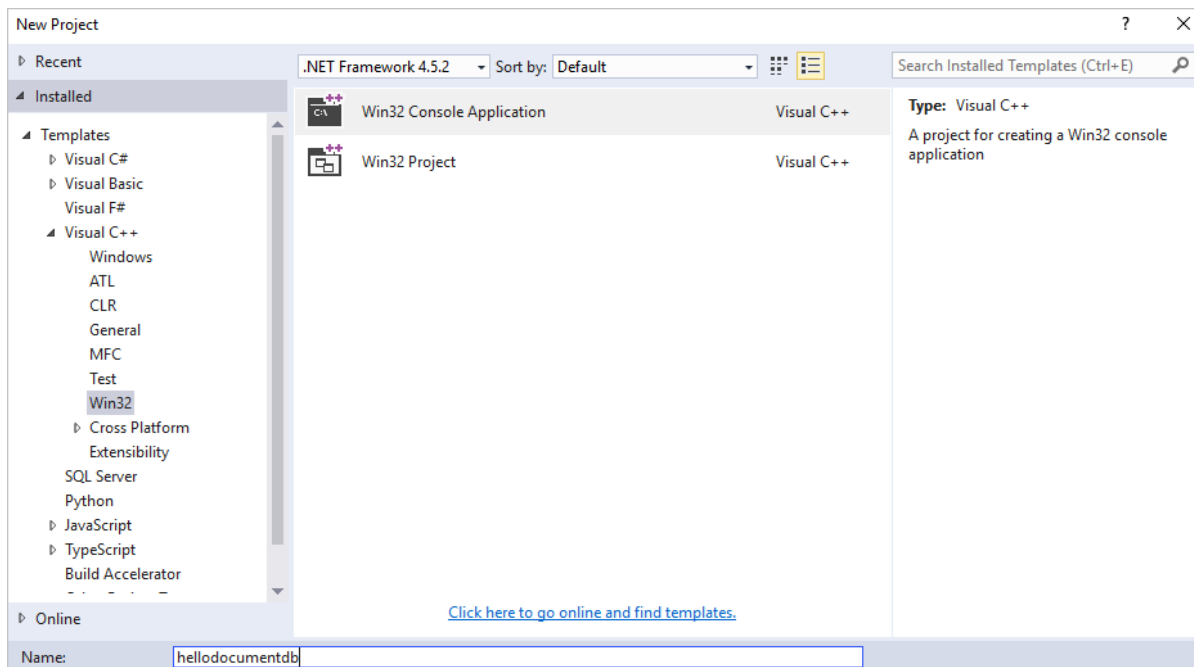
5. After the DocumentDB account is created, it is ready for use with the default settings. To review the default settings, click the **NoSQL (DocumentDB)** icon on the Jumpbar, click your new account, and then click **Default Consistency** in the Resource Menu.



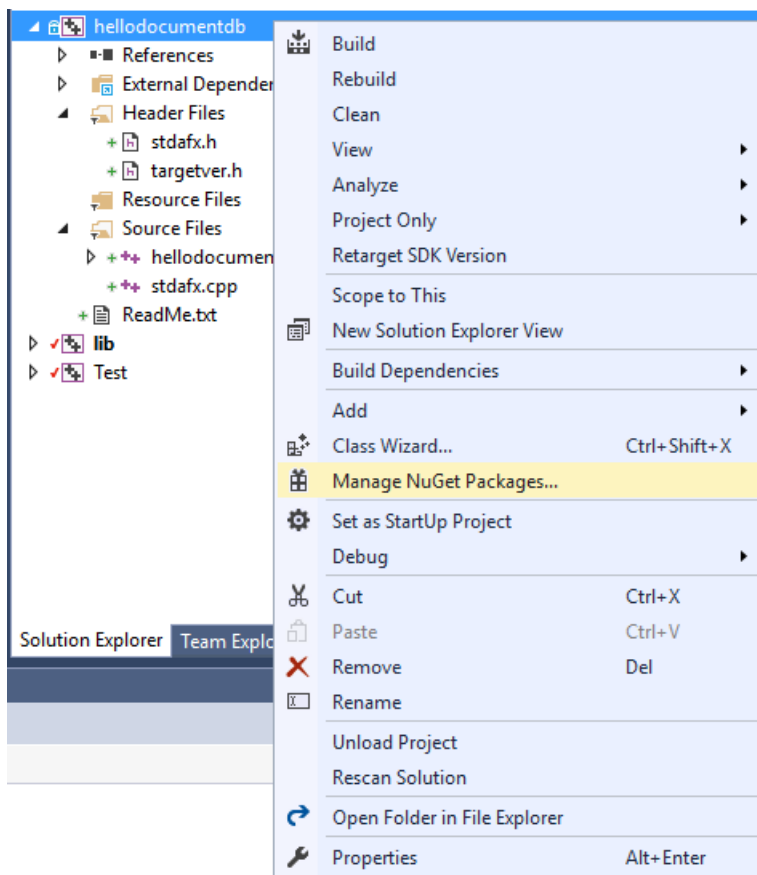
The default consistency of the DocumentDB account is set to **Session**. You can adjust the default consistency by clicking **Default Consistency** in the resource menu. To learn more about the consistency levels offered by DocumentDB, see [Consistency levels in DocumentDB](#).

Step 2: Set up your C++ application

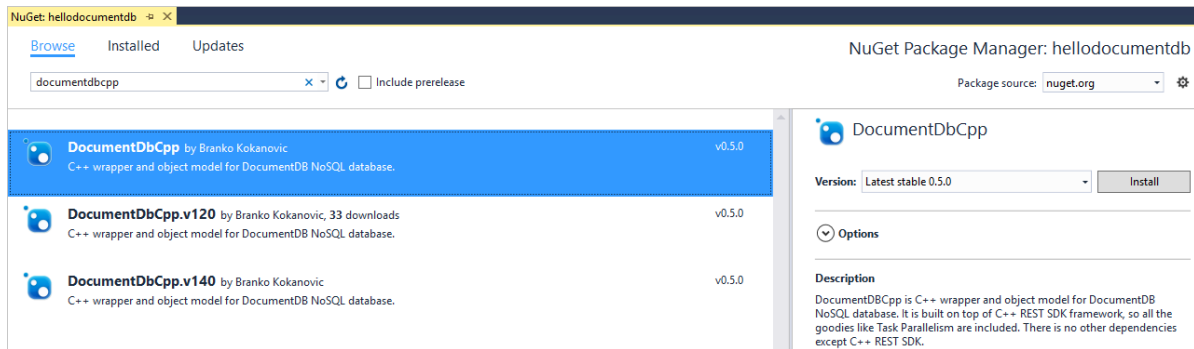
1. Open Visual Studio, and then on the **File** menu, click **New**, and then click **Project**.
2. In the **New Project** window, in the **Installed** pane, expand **Visual C++**, click **Win32**, and then click **Win32 Console Application**. Name the project `hellodocumentdb` and then click **OK**.



3. When the Win32 Application Wizard starts, click **Finish**.
4. Once the project has been created, open the NuGet package manager by right-clicking the **hellodocumentdb** project in **Solution Explorer** and clicking **Manage NuGet Packages**.



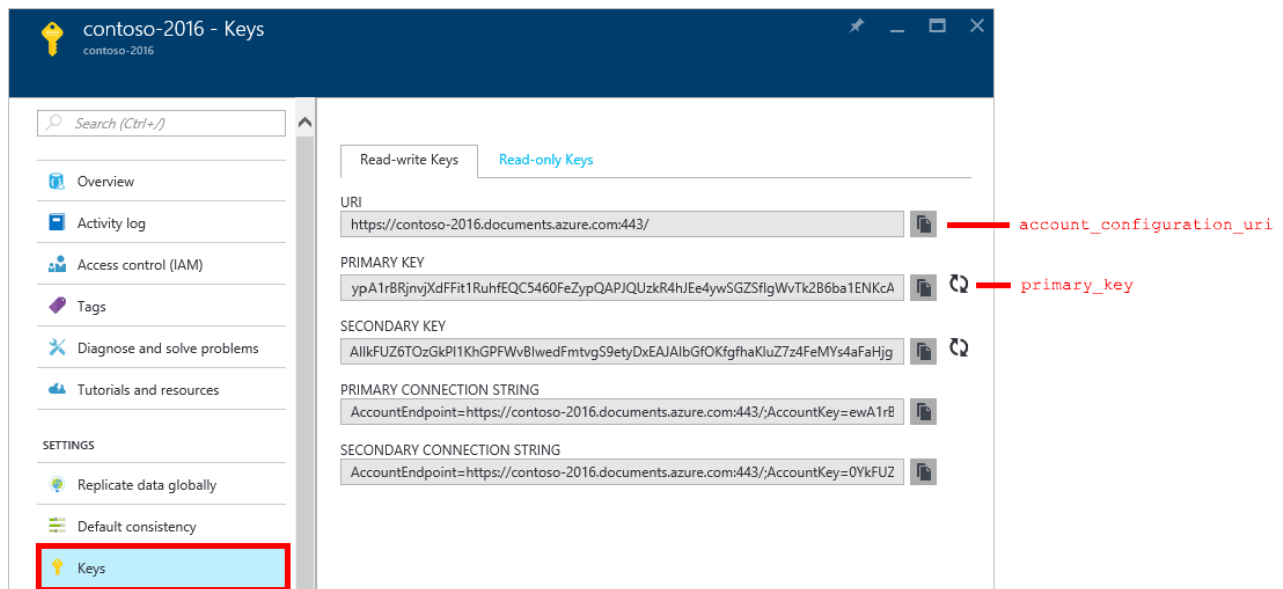
5. In the **NuGet: hellodocumentdb** tab, click **Browse**, and then search for *documentdbcpp*. In the results, select DocumentDbCPP, as shown in the following screenshot. This package installs references to C++ REST SDK, which is a dependency for the DocumentDbCPP.



Once the packages have been added to your project, we are all set to start writing some code.

Step 3: Copy connection details from Azure portal for your DocumentDB database

Bring up [Azure portal](#) and traverse to the NoSQL (DocumentDB) database account you created. We will need the URI and the primary key from Azure portal in the next step to establish a connection from our C++ code snippet.



Step 4: Connect to a DocumentDB account

1. Add the following headers and namespaces to your source code, after `#include "stdafx.h"`.

```
#include <cpprest/json>
#include <documentdbcpp\DocumentClient.h>
#include <documentdbcpp\exceptions.h>
#include <documentdbcpp\TriggerOperation.h>
#include <documentdbcpp\TriggerType.h>
using namespace documentdb;
using namespace std;
using namespace web::json;
```

2. Next add the following code to your main function and replace the account configuration and primary key to match your DocumentDB settings from step 3.

```
DocumentDBConfiguration conf (L"<account_configuration_uri>", L"<primary_key>");
DocumentClient client (conf);
```

Now that you have the code to initialize the documentdb client, let's take a look at working with DocumentDB resources.

Step 5: Create a C++ database and collection

Before we perform this step, let's go over how a database, collection and documents interact for those of you who are new to DocumentDB. A [database](#) is a logical container of document storage portioned across collections. A [collection](#) is a container of JSON documents and the associated JavaScript application logic. You can learn more about the DocumentDB hierarchical resource model and concepts in [DocumentDB hierarchical resource model and concepts](#).

To create a database and a corresponding collection add the following code to the end of your main function. This creates a database called 'FamilyRegistry' and a collection called 'FamilyCollection' using the client configuration you declared in the previous step.

```
try {
    shared_ptr<Database> db = client.CreateDatabase(L"FamilyRegistry");
    shared_ptr<Collection> coll = db->CreateCollection(L"FamilyCollection");
} catch (DocumentDBRuntimeExpection ex) {
    wcout << ex.message();
}
```

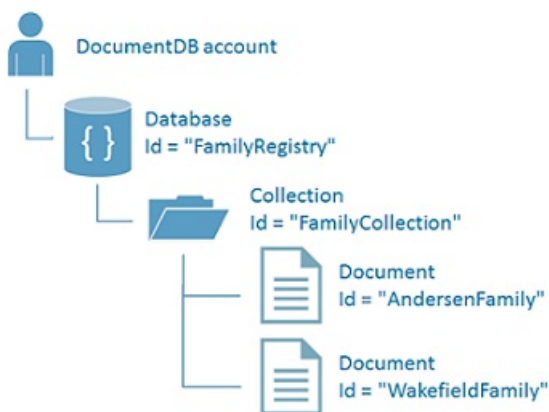
Step 6: Create a document

[Documents](#) are user-defined (arbitrary) JSON content. You can now insert a document into DocumentDB. You can create a document by copying the following code into the end of the main function.

```
try {
    value document_family;
    document_family[L"id"] = value::string(L"AndersenFamily");
    document_family[L"FirstName"] = value::string(L"Thomas");
    document_family[L"LastName"] = value::string(L"Andersen");
    shared_ptr<Document> doc = coll->CreateDocumentAsync(document_family).get();

    document_family[L"id"] = value::string(L"WakefieldFamily");
    document_family[L"FirstName"] = value::string(L"Lucy");
    document_family[L"LastName"] = value::string(L"Wakefield");
    doc = coll->CreateDocumentAsync(document_family).get();
} catch (ResourceAlreadyExistsException ex) {
    wcout << ex.message();
}
```

To summarize, this code creates a DocumentDB database, collection, and documents, which you can query in Document Explorer in Azure portal.



Step 7: Query DocumentDB resources

DocumentDB supports [rich queries](#) against JSON documents stored in each collection. The following sample code

shows a query made using DocumentDB SQL syntax that you can run against the documents we created in the previous step.

The function takes in as arguments the unique identifier or resource id for the database and the collection along with the document client. Add this code before main function.

```
void executesimplequery(const DocumentClient &client,
                        const wstring dbresourceid,
                        const wstring collresourceid) {
    try {
        client.GetDatabase(dbresourceid).get();
        shared_ptr<Database> db = client.GetDatabase(dbresourceid);
        shared_ptr<Collection> coll = db->GetCollection(collresourceid);
        wstring coll_name = coll->id();
        shared_ptr<DocumentIterator> iter =
            coll->QueryDocumentsAsync(wstring(L"SELECT * FROM " + coll_name)).get();
        wcout << "\n\nQuerying collection:";
        while (iter->HasMore()) {
            shared_ptr<Document> doc = iter->Next();
            wstring doc_name = doc->id();
            wcout << "\n\t" << doc_name << "\n";
            wcout << "\t"
                << "[{"FirstName\":"
                << doc->payload().at(U("FirstName")).as_string()
                << ",\LastName\":" << doc->payload().at(U("LastName")).as_string()
                << "}]";
        }
    } catch (DocumentDBRuntimeExcepion ex) {
        wcout << ex.message();
    }
}
```

Step 8: Replace a document

DocumentDB supports replacing JSON documents, as demonstrated in the following code. Add this code after the `executesimplequery` function.

```
void replacedocument(const DocumentClient &client, const wstring dbresourceid,
                     const wstring collresourceid,
                     const wstring docresourceid) {
    try {
        client.GetDatabase(dbresourceid).get();
        shared_ptr<Database> db = client.GetDatabase(dbresourceid);
        shared_ptr<Collection> coll = db->GetCollection(collresourceid);
        value newdoc;
        newdoc[L"id"] = value::string(L"WakefieldFamily");
        newdoc[L"FirstName"] = value::string(L"Lucy");
        newdoc[L"LastName"] = value::string(L"Smith Wakefield");
        coll->ReplaceDocument(docresourceid, newdoc);
    } catch (DocumentDBRuntimeExcepion ex) {
        throw;
    }
}
```

Step 9: Delete a document

DocumentDB supports deleting JSON documents, you can do so by copy and pasting the following code after the `replacedocument` function.

```

void deletedocument(const DocumentClient &client, const wstring dbresourceid,
                    const wstring collresourceid, const wstring docresourceid) {
    try {
        client.GetDatabase(dbresourceid).get();
        shared_ptr<Database> db = client.GetDatabase(dbresourceid);
        shared_ptr<Collection> coll = db->GetCollection(collresourceid);
        coll->DeleteDocumentAsync(docresourceid).get();
    } catch (DocumentDBRuntimeExpection ex) {
        wcout << ex.message();
    }
}

```

Step 10: Delete a database

Deleting the created database removes the database and all child resources (collections, documents, etc.).

Copy and paste the following code snippet (function cleanup) after the deletedocument function to remove the database and all the child resources.

```

void deletedb(const DocumentClient &client, const wstring dbresourceid) {
    try {
        client.DeleteDatabase(dbresourceid);
    } catch (DocumentDBRuntimeExpection ex) {
        wcout << ex.message();
    }
}

```

Step 11: Run your C++ application all together!

We have now added code to create, query, modify, and delete different DocumentDB resources. Let us now wire this up by adding calls to these different functions from our main function in hellodocumentdb.cpp along with some diagnostic messages.

You can do so by replacing the main function of your application with the following code. This writes over the account_configuration_uri and primary_key you copied into the code in Step 3, so save that line or copy the values in again from the portal.

```

int main() {
    try {
        // Start by defining your account's configuration
        DocumentDBConfiguration conf (L"<account_configuration_uri>", L"<primary_key>");
        // Create your client
        DocumentClient client(conf);
        // Create a new database
        try {
            shared_ptr<Database> db = client.CreateDatabase(L"FamilyDB");
            wcout << "\nCreating database:\n" << db->id();
            // Create a collection inside database
            shared_ptr<Collection> coll = db->CreateCollection(L"FamilyColl");
            wcout << "\n\nCreating collection:\n" << coll->id();
            value document_family;
            document_family[L"id"] = value::string(L"AndersenFamily");
            document_family[L"FirstName"] = value::string(L"Thomas");
            document_family[L"LastName"] = value::string(L"Andersen");
            shared_ptr<Document> doc =
                coll->CreateDocumentAsync(document_family).get();
            wcout << "\n\nCreating document:\n" << doc->id();
            document_family[L"id"] = value::string(L"WakefieldFamily");
            document_family[L"FirstName"] = value::string(L"Lucy");
            document_family[L"LastName"] = value::string(L"Wakefield");
            doc = coll->CreateDocumentAsync(document_family).get();
            wcout << "\n\nCreating document:\n" << doc->id();
            executesimplequery(client, db->resource_id(), coll->resource_id());
            replacedocument(client, db->resource_id(), coll->resource_id(),
                doc->resource_id());
            wcout << "\n\nReplaced document:\n" << doc->id();
            executesimplequery(client, db->resource_id(), coll->resource_id());
            deletedocument(client, db->resource_id(), coll->resource_id(),
                doc->resource_id());
            wcout << "\n\nDeleted document:\n" << doc->id();
            deletedb(client, db->resource_id());
            wcout << "\n\nDeleted db:\n" << db->id();
            cin.get();
        }
        catch (ResourceAlreadyExistsException ex) {
            wcout << ex.message();
        }
    }
    catch (DocumentDBRuntimeExcpetion ex) {
        wcout << ex.message();
    }
    cin.get();
}

```

You should now be able to build and run your code in Visual Studio by pressing F5 or alternatively in the terminal window by locating the application and running the executable.

You should see the output of your get started app. The output should match the following screenshot.

```
Creating database:
FamilyDB

Creating collection:
FamilyColl

Creating document:
AndersenFamily

Creating document:
WakefieldFamily

Querying collection:
AndersenFamily
[{"FirstName":Thomas,"LastName":Andersen}]
WakefieldFamily
[{"FirstName":Lucy,"LastName":Wakefield}]

Replacing document:
WakefieldFamily

Querying collection:
AndersenFamily
[{"FirstName":Thomas,"LastName":Andersen}]
WakefieldFamily
[{"FirstName":Lucy,"LastName":Smith Wakefield}]

Deleted document:
WakefieldFamily

Deleted db:
FamilyDB
```

Congratulations! You've completed the C++ tutorial and have your first DocumentDB console application!

Get the complete C++ tutorial solution

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

- [DocumentDB account](#).
- The [GetStarted](#) solution available on GitHub.

Next steps

- Learn how to [monitor a DocumentDB account](#).
- Run queries against our sample dataset in the [Query Playground](#).
- Learn more about the programming model in the Develop section of the [DocumentDB documentation page](#).

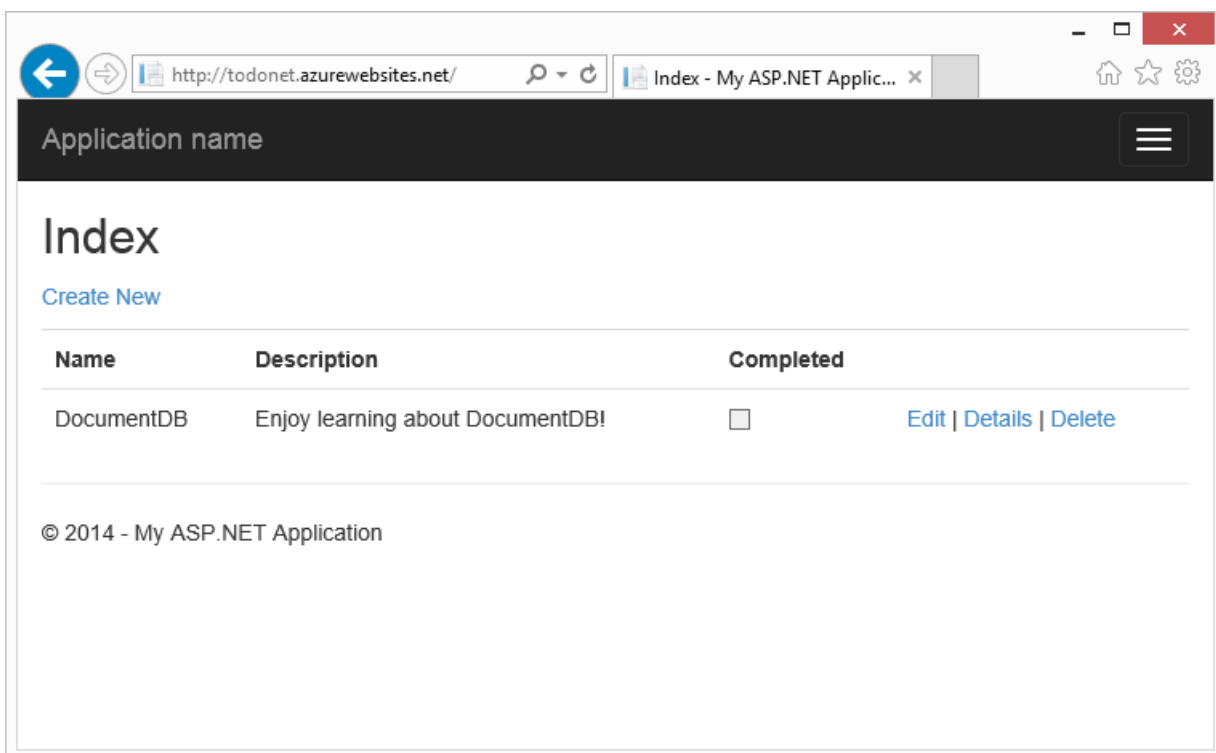
ASP.NET MVC Tutorial: Web application development with DocumentDB

11/22/2016 • 19 min to read • [Edit on GitHub](#)

Contributors

Syam Nair • mimig • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Andrew Liu • joescars • Loren Paulsen • arramac • Ryan CrawCour • v-aljenk • Dan Friedman • Dene Hager

To highlight how you can efficiently leverage Azure DocumentDB to store and query JSON documents, this article provides an end-to-end walk-through showing you how to build a todo app using Azure DocumentDB. The tasks will be stored as JSON documents in Azure DocumentDB.



This walk-through shows you how to use the DocumentDB service provided by Azure to store and access data from an ASP.NET MVC web application hosted on Azure. If you're looking for a tutorial that focuses only on DocumentDB, and not the ASP.NET MVC components, see [Build a DocumentDB C# console application](#).

TIP

This tutorial assumes that you have prior experience using ASP.NET MVC and Azure Websites. If you are new to ASP.NET or the [prerequisite tools](#), we recommend downloading the complete sample project from [GitHub](#) and following the instructions in this sample. Once you have it built, you can review this article to gain insight on the code in the context of the project.

Prerequisites for this database tutorial

Before following the instructions in this article, you should ensure that you have the following:

- An active Azure account. If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see [Azure Free Trial](#)

OR

A local installation of the [Azure DocumentDB Emulator](#).

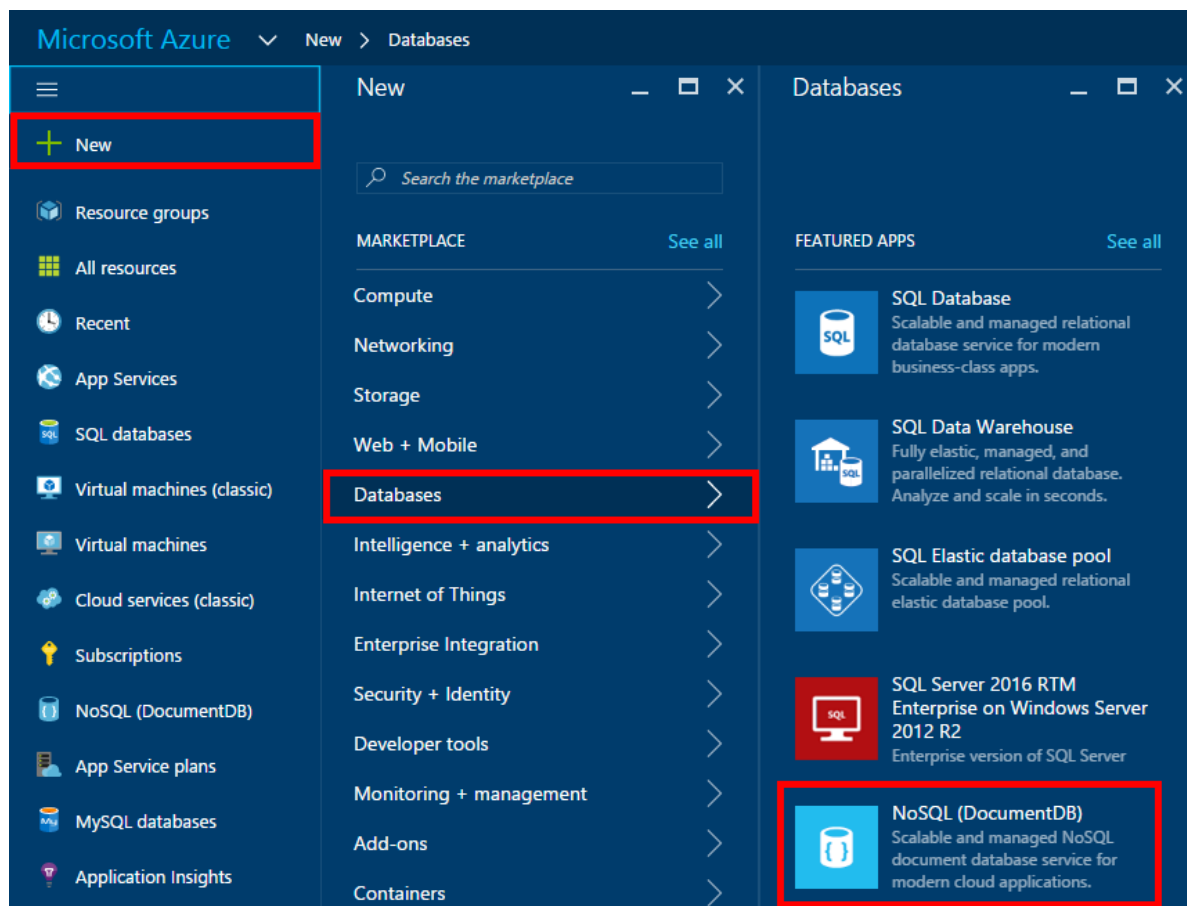
- [Visual Studio 2015](#) or Visual Studio 2013 Update 4 or higher. If using Visual Studio 2013, you will need to install the [Microsoft.Net.Compilers nuget package](#) to add support for C# 6.0.
- Azure SDK for .NET version 2.5.1 or higher, available through the [Microsoft Web Platform Installer](#).

All the screen shots in this article have been taken using Visual Studio 2013 with Update 4 applied and the Azure SDK for .NET version 2.5.1. If your system is configured with different versions it is possible that your screens and options won't match entirely, but if you meet the above prerequisites this solution should work.

Step 1: Create a DocumentDB database account

Let's start by creating a DocumentDB account. If you already have an account or if you are using the DocumentDB Emulator for this tutorial, you can skip to [Create a new ASP.NET MVC application](#).

1. Sign in to the [Azure portal](#).
2. In the Jumpbar, click **New**, click **Databases**, and then click **NoSQL (DocumentDB)**.



3. In the **New** account blade, specify the desired configuration for the DocumentDB account.

NoSQL (DocumentDB)
New account

* ID
contosoacct ✓
documents.azure.com

NoSQL API ⓘ
DocumentDB MongoDB

* Subscription
Visual Studio Ultimate with MSDN ▼

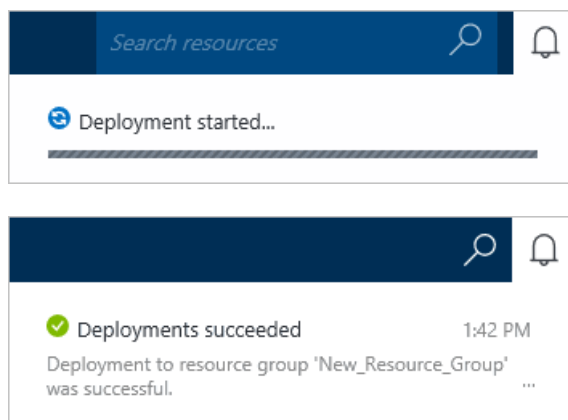
* Resource Group ⓘ
☒ Create new ☐ Use existing
contosoacct ✓

* Location
West US ▼

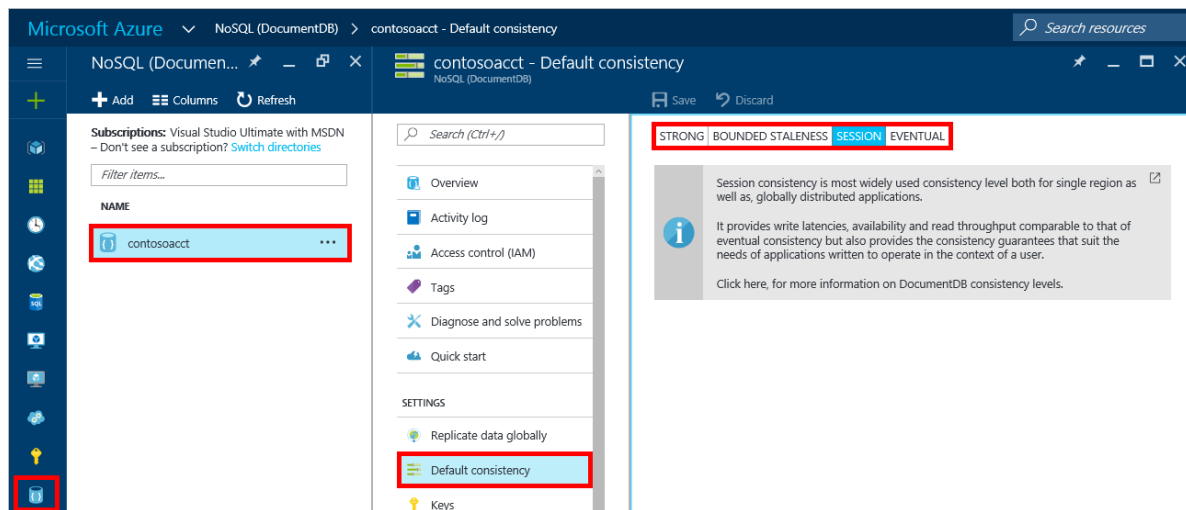
☐ Pin to dashboard

Create Automation options

- In the **ID** box, enter a name to identify the DocumentDB account. When the ID is validated, a green check mark appears in the ID box. The ID value becomes the host name within the URI. The ID may contain only lowercase letters, numbers, and the '-' character, and must be between 3 and 50 characters. Note that *documents.azure.com* is appended to the endpoint name you choose, the result of which becomes your DocumentDB account endpoint.
 - In the **NoSQL API** box, select **DocumentDB**.
 - For **Subscription**, select the Azure subscription that you want to use for the DocumentDB account. If your account has only one subscription, that account is selected by default.
 - In **Resource Group**, select or create a resource group for your DocumentDB account. By default, a new resource group is created. For more information, see [Using the Azure portal to manage your Azure resources](#).
 - Use **Location** to specify the geographic location in which to host your DocumentDB account.
4. Once the new DocumentDB account options are configured, click **Create**. To check the status of the deployment, check the Notifications hub.

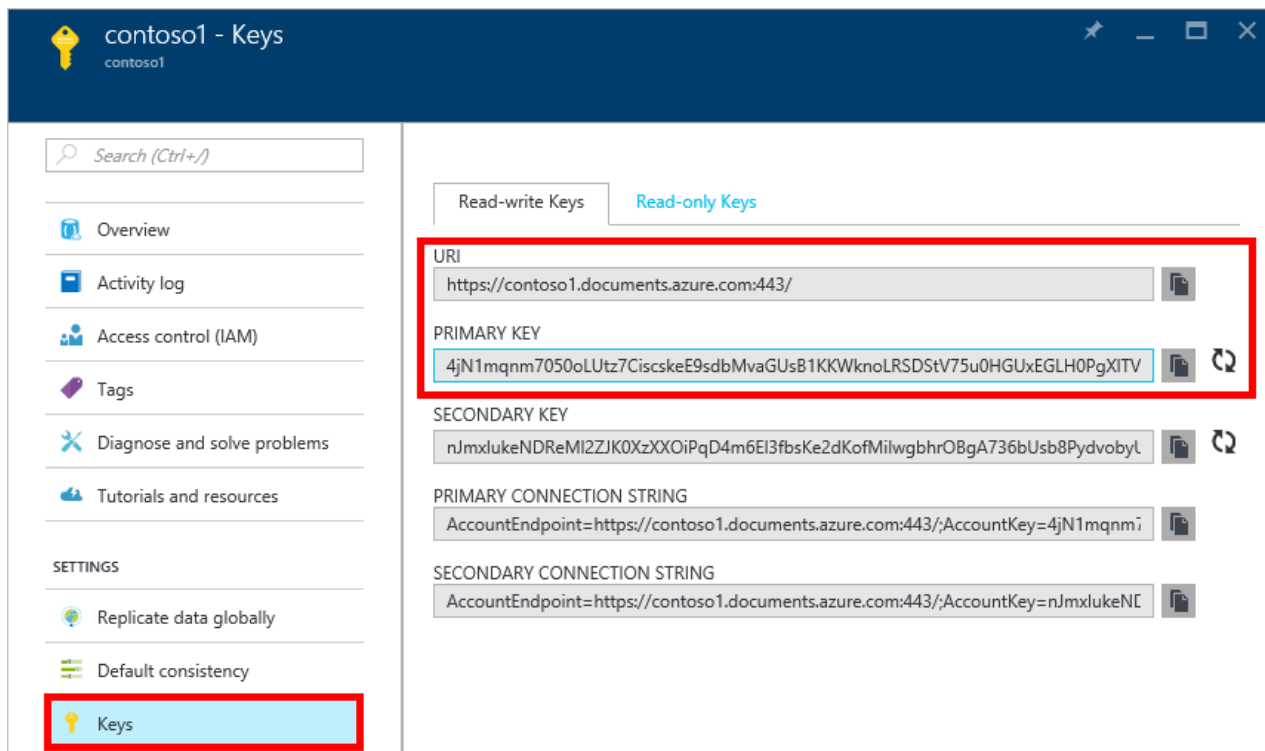


5. After the DocumentDB account is created, it is ready for use with the default settings. To review the default settings, click the **NoSQL (DocumentDB)** icon on the Jumpbar, click your new account, and then click **Default Consistency** in the Resource Menu.



The default consistency of the DocumentDB account is set to **Session**. You can adjust the default consistency by clicking **Default Consistency** in the resource menu. To learn more about the consistency levels offered by DocumentDB, see [Consistency levels in DocumentDB](#).

Now navigate to the DocumentDB account blade, and click **Keys**, as we will use these values in the web application we create next.



We will now walk through how to create a new ASP.NET MVC application from the ground-up.

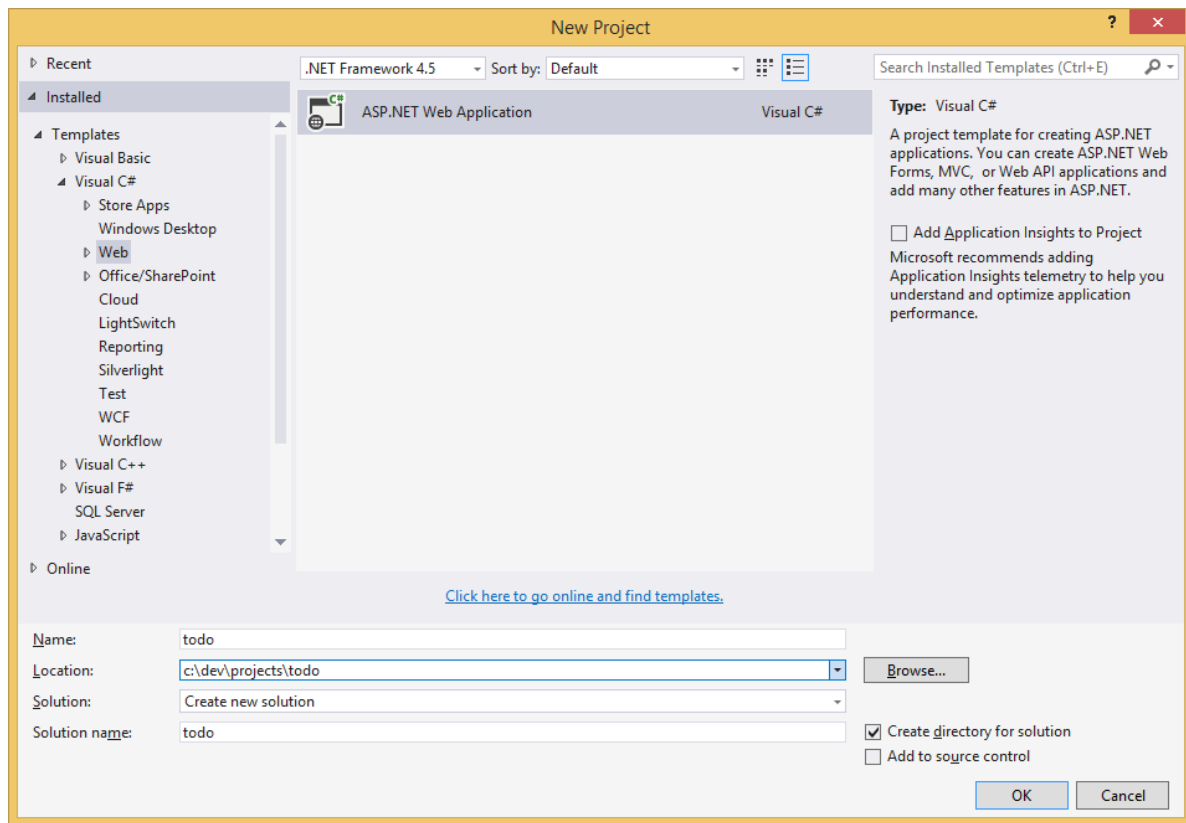
Step 2: Create a new ASP.NET MVC application

Now that you have an account, let's create our new ASP.NET project.

1. In Visual Studio, on the **File** menu, point to **New**, and then click **Project**.

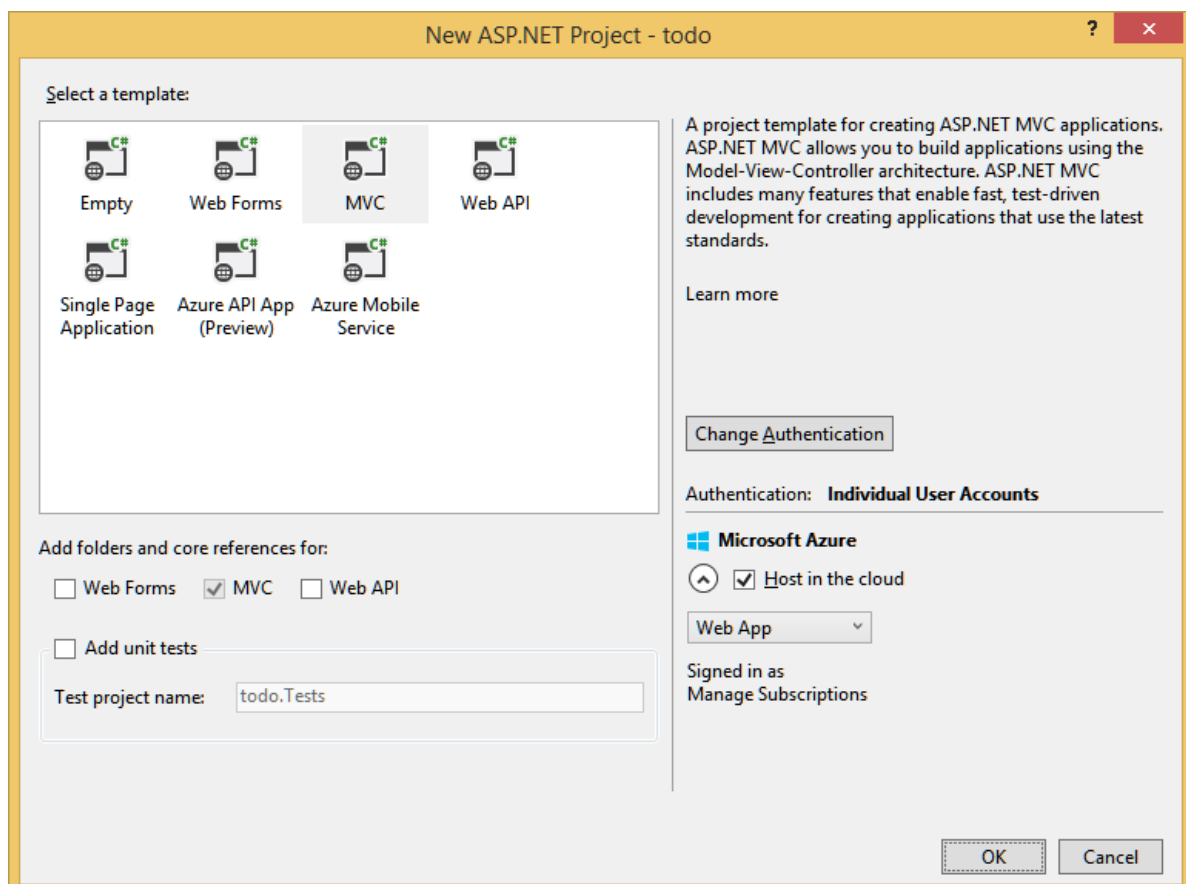
The ****New Project**** dialog box appears.

2. In the **Project types** pane, expand **Templates**, **Visual C#**, **Web**, and then select **ASP.NET Web Application**.



3. In the **Name** box, type the name of the project. This tutorial uses the name "todo". If you choose to use something other than this, then wherever this tutorial talks about the todo namespace, you need to adjust the provided code samples to use whatever you named your application.
4. Click **Browse** to navigate to the folder where you would like to create the project, and then click **OK**.

The **New ASP.NET Project** dialog box appears.



5. In the templates pane, select **MVC**.

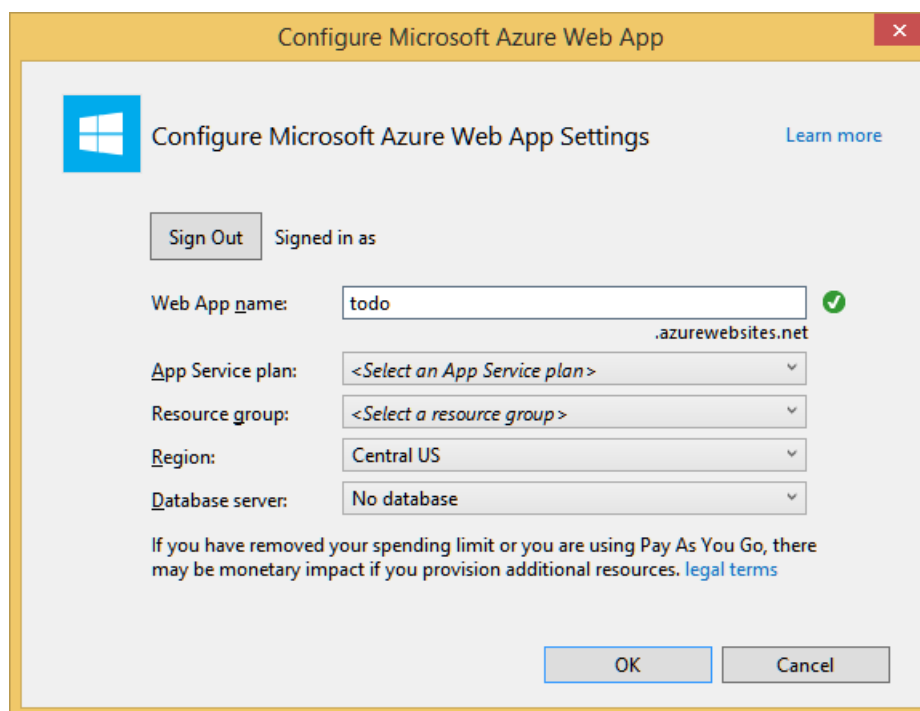
6. If you plan on hosting your application in Azure then select **Host in the cloud** on the lower right to have Azure host the application. We've selected to host in the cloud, and to run the application hosted in an Azure Website. Selecting this option will preprovision an Azure Website for you and make life a lot easier when it comes time to deploy the final working application. If you want to host this elsewhere or don't want to configure Azure upfront, then just clear **Host in the Cloud**.
7. Click **OK** and let Visual Studio do its thing around scaffolding the empty ASP.NET MVC template.

If you receive the error "An error occurred while processing your request" see the [Troubleshooting](#) section.

8. If you chose to host this in the cloud you will see at least one additional screen asking you to login to your Azure account and provide some values for your new website. Supply all the additional values and continue.

I haven't chosen a "Database server" here because we're not using an Azure SQL Database Server here, we're going to be creating a new Azure DocumentDB account later on in the Azure Portal.

For more information about choosing an **App Service plan** and **Resource group**, see [Azure App Service plans in-depth overview](#).



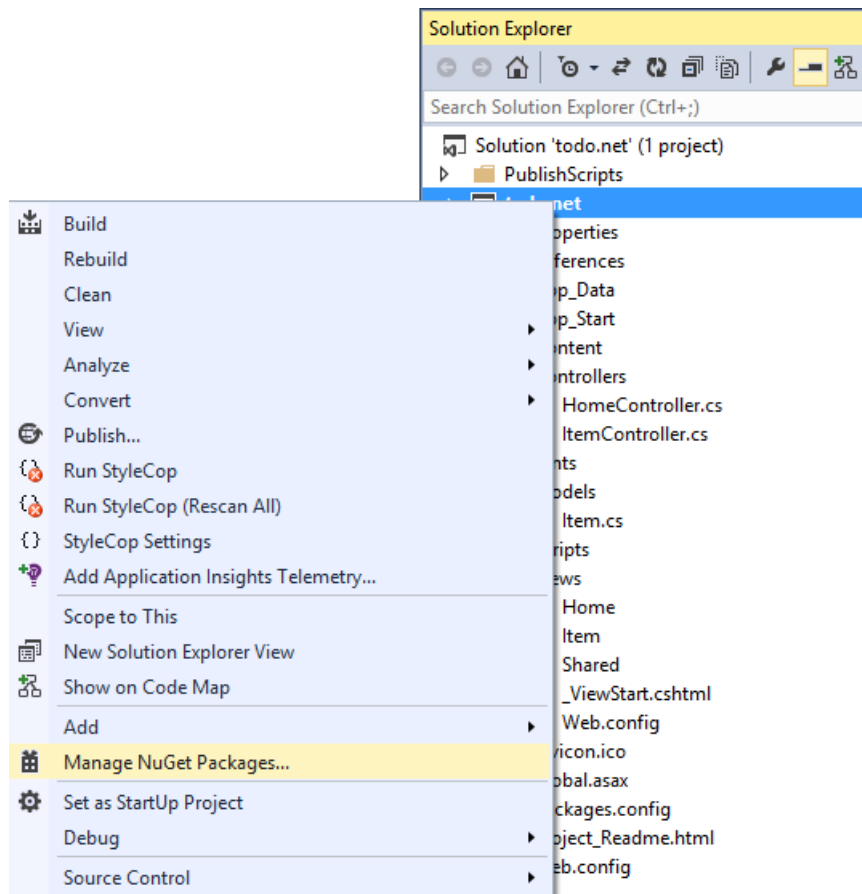
9. Once Visual Studio has finished creating the boilerplate MVC application you have an empty ASP.NET application that you can run locally.

We'll skip running the project locally because I'm sure we've all seen the ASP.NET "Hello World" application. Let's go straight to adding DocumentDB to this project and building our application.

Step 3: Add DocumentDB to your MVC web application project

Now that we have most of the ASP.NET MVC plumbing that we need for this solution, let's get to the real purpose of this tutorial, adding Azure DocumentDB to our MVC web application.

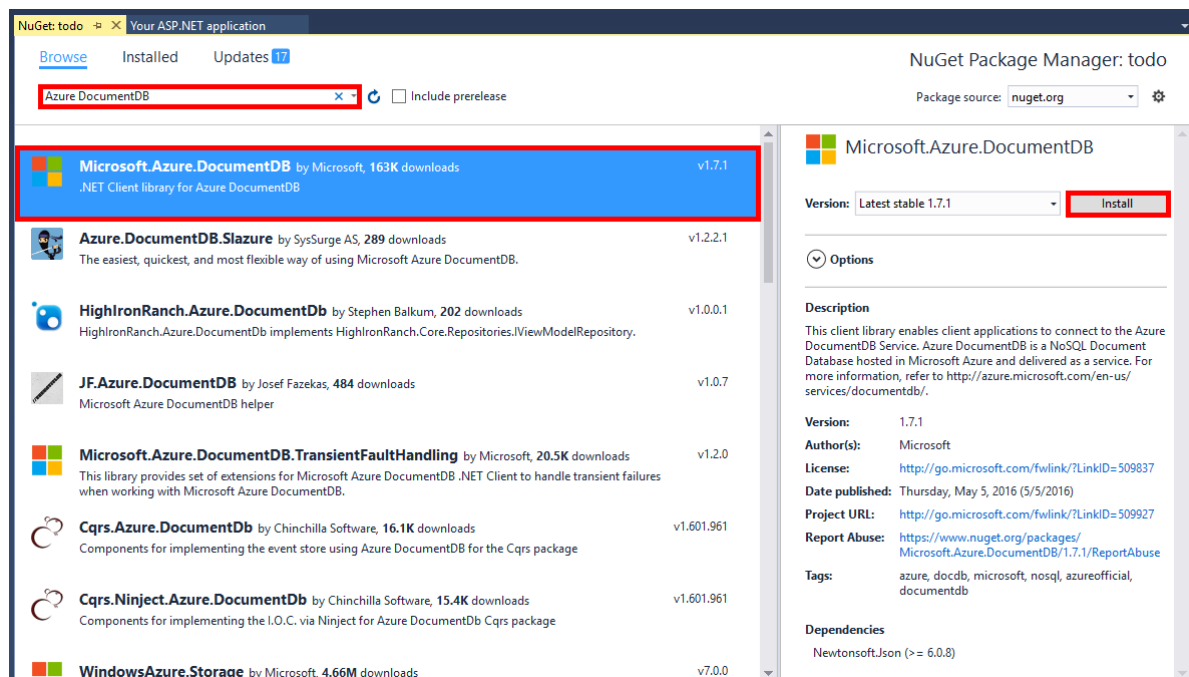
1. The DocumentDB .NET SDK is packaged and distributed as a NuGet package. To get the NuGet package in Visual Studio, use the NuGet package manager in Visual Studio by right-clicking on the project in **Solution Explorer** and then clicking **Manage NuGet Packages**.



The **Manage NuGet Packages** dialog box appears.

2. In the NuGet Browse box, type *Azure DocumentDB*.

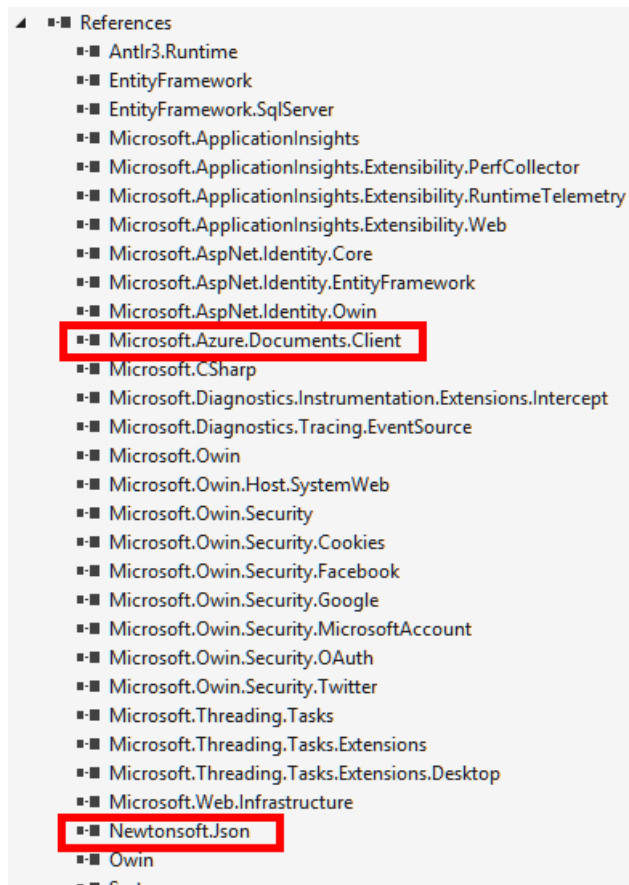
From the results, install the **Microsoft Azure DocumentDB Client Library** package. This will download and install the DocumentDB package as well as all dependencies, like Newtonsoft.Json. Click **OK** in the **Preview** window, and **I Accept** in the **License Acceptance** window to complete the install.



Alternatively you can use the Package Manager Console to install the package. To do so, on the **Tools** menu, click **NuGet Package Manager**, and then click **Package Manager Console**. At the prompt, type the following.

```
Install-Package Microsoft.Azure.DocumentDB
```

- Once the package is installed, your Visual Studio solution should resemble the following with two new references added, Microsoft.Azure.Documents.Client and Newtonsoft.Json.



Step 4: Set up the ASP.NET MVC application

Now let's add the models, views, and controllers to this MVC application:

- [Add a model.](#)
- [Add a controller.](#)
- [Add views.](#)

Add a JSON data model

Let's begin by creating the **M** in MVC, the model.

- In **Solution Explorer**, right-click the **Models** folder, click **Add**, and then click **Class**.

The **Add New Item** dialog box appears.

- Name your new class **Item.cs** and click **Add**.
- In this new **Item.cs** file, add the following after the last *using statement*.

```
using Newtonsoft.Json;
```

- Now replace this code

```
public class Item
{
}
```

with the following code.

```
public class Item
{
    [JsonProperty(PropertyName = "id")]
    public string Id { get; set; }

    [JsonProperty(PropertyName = "name")]
    public string Name { get; set; }

    [JsonProperty(PropertyName = "description")]
    public string Description { get; set; }

    [JsonProperty(PropertyName = "isComplete")]
    public bool Completed { get; set; }
}
```

All data in DocumentDB is passed over the wire and stored as JSON. To control the way your objects are serialized/deserialized by JSON.NET you can use the **JsonProperty** attribute as demonstrated in the **Item** class we just created. You don't **have** to do this but I want to ensure that my properties follow the JSON camelCase naming conventions.

Not only can you control the format of the property name when it goes into JSON, but you can entirely rename your .NET properties like I did with the **Description** property.

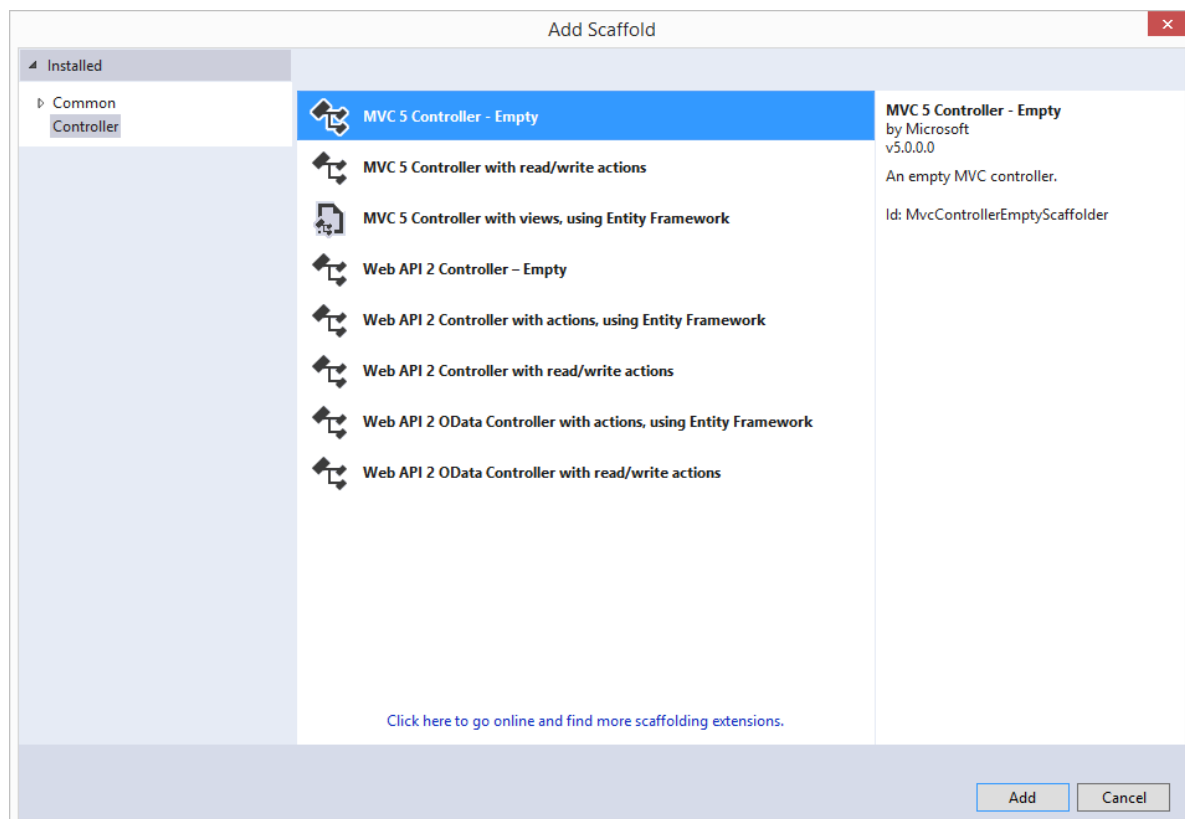
Add a controller

That takes care of the **M**, now let's create the **C** in MVC, a controller class.

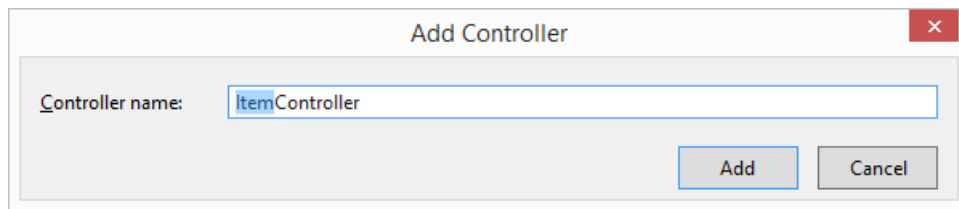
1. In **Solution Explorer**, right-click the **Controllers** folder, click **Add**, and then click **Controller**.

The **Add Scaffold** dialog box appears.

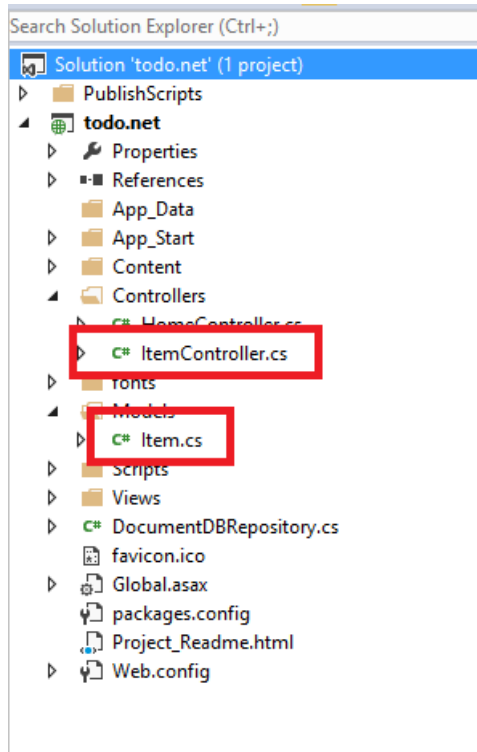
2. Select **MVC 5 Controller - Empty** and then click **Add**.



3. Name your new Controller, **ItemController**.



Once the file is created, your Visual Studio solution should resemble the following with the new ItemController.cs file in **Solution Explorer**. The new Item.cs file created earlier is also shown.



You can close ItemController.cs, we'll come back to it later.

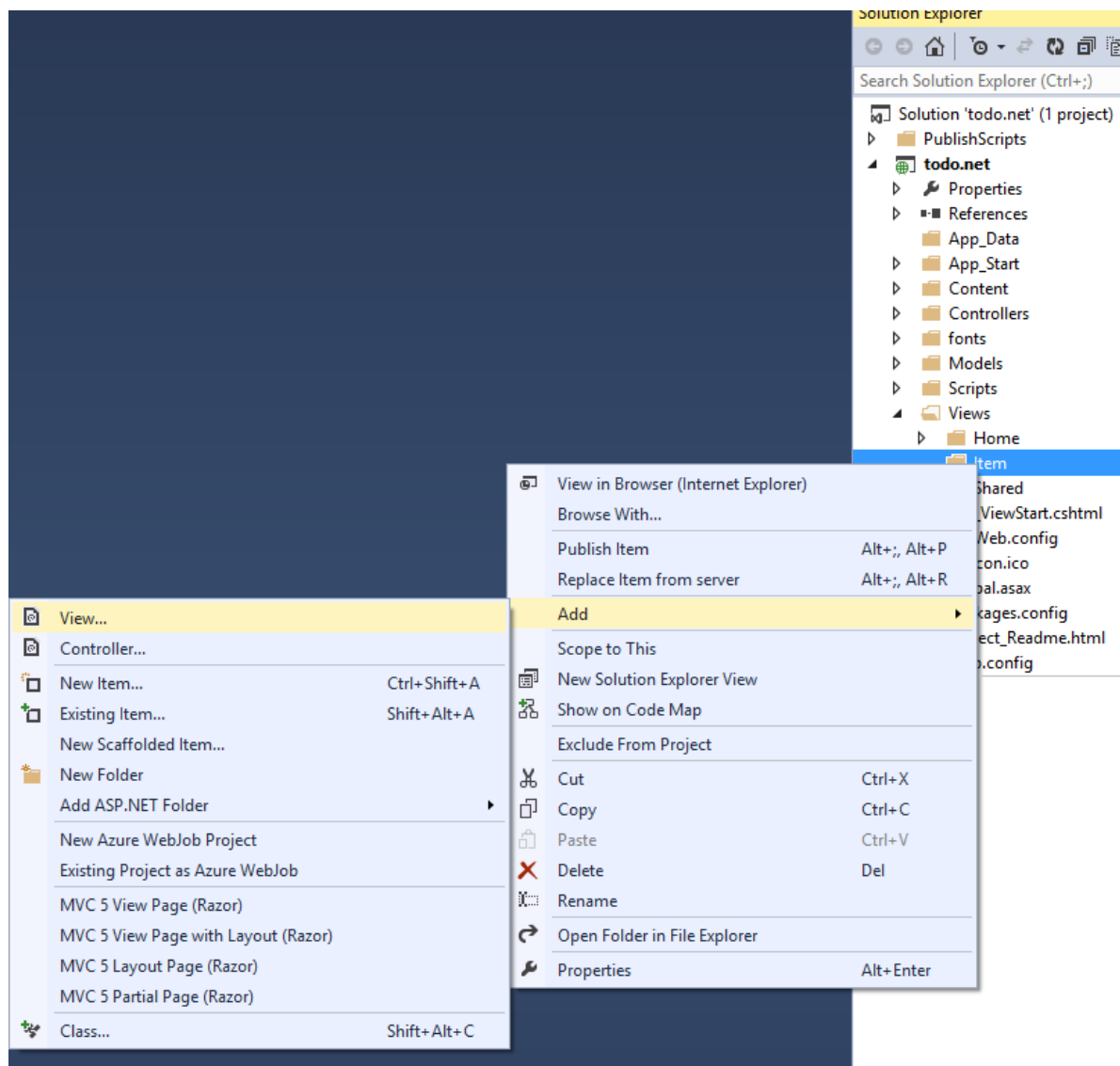
Add views

Now, let's create the V in MVC, the views:

- [Add an Item Index view.](#)
- [Add a New Item view.](#)
- [Add an Edit Item view.](#)

Add an Item Index view

1. In **Solution Explorer**, expand the **Views** folder, right-click the empty **Item** folder that Visual Studio created for you when you added the **ItemController** earlier, click **Add**, and then click **View**.



2. In the **Add View** dialog box, do the following:

- In the **View name** box, type *Index*.
- In the **Template** box, select *List*.
- In the **Model class** box, select *Item (todo.Models)*.
- Leave the **Data context class** box empty.
- In the layout page box, type *~/Views/Shared/_Layout.cshtml*.

Add View

View name:

Template:

Model class:

Data context class:

Options:

☐ Create as a partial view

☒ Reference script libraries

☒ Use a layout page:

...

(Leave empty if it is set in a Razor _viewstart file)

3. Once all these values are set, click **Add** and let Visual Studio create a new template view. Once it is done, it will open the cshtml file that was created. We can close that file in Visual Studio as we will come back to it later.

Add a New Item view

Similar to how we created an **Item Index** view, we will now create a new view for creating new **Items**.

1. In **Solution Explorer**, right-click the **Item** folder again, click **Add**, and then click **View**.
2. In the **Add View** dialog box, do the following:
 - In the **View name** box, type *Create*.
 - In the **Template** box, select *Create*.
 - In the **Model class** box, select *Item (todo.Models)*.
 - Leave the **Data context class** box empty.
 - In the layout page box, type *~/Views/Shared/_Layout.cshtml*.
 - Click **Add**.

Add an Edit Item view

And finally, add one last view for editing an **Item** in the same way as before.

1. In **Solution Explorer**, right-click the **Item** folder again, click **Add**, and then click **View**.
2. In the **Add View** dialog box, do the following:
 - In the **View name** box, type *Edit*.
 - In the **Template** box, select *Edit*.
 - In the **Model class** box, select *Item (todo.Models)*.
 - Leave the **Data context class** box empty.
 - In the layout page box, type *~/Views/Shared/_Layout.cshtml*.
 - Click **Add**.

Once this is done, close all the cshtml documents in Visual Studio as we will return to these views later.

Step 5: Wiring up DocumentDB

Now that the standard MVC stuff is taken care of, let's turn to adding the code for DocumentDB.

In this section, we'll add code to handle the following:

- [Listing incomplete Items](#).
- [Adding Items](#).
- [Editing Items](#).

Listing incomplete Items in your MVC web application

The first thing to do here is add a class that contains all the logic to connect to and use DocumentDB. For this tutorial we'll encapsulate all this logic in to a repository class called **DocumentDBRepository**.

1. In **Solution Explorer**, right-click on the project, click **Add**, and then click **Class**. Name the new class **DocumentDBRepository** and click **Add**.
2. In the newly created **DocumentDBRepository** class and add the following *using statements* above the *namespace* declaration

```
using Microsoft.Azure.Documents;  
using Microsoft.Azure.Documents.Client;  
using Microsoft.Azure.Documents.Linq;  
using System.Configuration;  
using System.Linq.Expressions;  
using System.Threading.Tasks;
```

Now replace this code

```
public class DocumentDBRepository  
{  
}
```

with the following code.

```

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 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;
            }
        }
    }
}

```

TIP

When creating a new DocumentCollection you can supply an optional RequestOptions parameter of OfferType, which allows you to specify the performance level of the new collection. If this parameter is not passed the default offer type will be used. For more on DocumentDB offer types please refer to [DocumentDB Performance Levels](#)

3. We're reading some values from configuration, so open the **Web.config** file of your application and add

the following lines under the `<AppSettings>` section.

```
<add key="endpoint" value="enter the URI from the Keys blade of the Azure Portal"/>
<add key="authKey" value="enter the PRIMARY KEY, or the SECONDARY KEY, from the Keys blade of the Azure Portal"/>
<add key="database" value="ToDoList"/>
<add key="collection" value="Items"/>
```

- Now, update the values for *endpoint* and *authKey* using the Keys blade of the Azure Portal. Use the **URI** from the Keys blade as the value of the endpoint setting, and use the **PRIMARY KEY**, or **SECONDARY KEY** from the Keys blade as the value of the authKey setting.

That takes care of wiring up the DocumentDB repository, now let's add our application logic.

- The first thing we want to be able to do with a todo list application is to display the incomplete items. Copy and paste the following code snippet anywhere within the **DocumentDBRepository** class.

```
public static async Task<IEnumerable<T>> GetItemsAsync(Expression<Func<T, bool>> predicate)
{
    IDocumentQuery<T> query = client.CreateDocumentQuery<T>(
        UriFactory.CreateDocumentCollectionUri(DatabaseId, CollectionId))
        .Where(predicate)
        .AsDocumentQuery();

    List<T> results = new List<T>();
    while (query.HasMoreResults)
    {
        results.AddRange(await query.ExecuteNextAsync<T>());
    }

    return results;
}
```

- Open the **ItemController** we added earlier and add the following *using statements* above the namespace declaration.

```
using System.Net;
using System.Threading.Tasks;
using todo.Models;
```

If your project is not named "todo", then you need to update using "todo.Models"; to reflect the name of your project.

Now replace this code

```
//GET: Item
public ActionResult Index()
{
    return View();
}
```

with the following code.

```
[ActionName("Index")]
public async Task<ActionResult> IndexAsync()
{
    var items = await DocumentDBRepository<Item>.GetItemsAsync(d => !d.Completed);
    return View(items);
}
```

7. Open **Global.asax.cs** and add the following line to the **Application_Start** method

```
DocumentDBRepository<todo.Models.Item>.Initialize();
```

At this point your solution should be able to build without any errors.

If you ran the application now, you would go to the **HomeController** and the **Index** view of that controller. This is the default behavior for the MVC template project we chose at the start but we don't want that! Let's change the routing on this MVC application to alter this behavior.

Open **App_Start\RouteConfig.cs** and locate the line starting with "defaults:" and change it to resemble the following.

```
defaults: new { controller = "Item", action = "Index", id = UrlParameter.Optional }
```

This now tells ASP.NET MVC that if you have not specified a value in the URL to control the routing behavior that instead of **Home**, use **Item** as the controller and user **Index** as the view.

Now if you run the application, it will call into your **ItemController** which will call in to the repository class and use the **GetItems** method to return all the incomplete items to the **Views\Item\Index** view.

If you build and run this project now, you should now see something that looks this.

Index		
Create New		
Name	Description	Completed
© 2014 - Azure DocumentDB		

Adding Items

Let's put some items into our database so we have something more than an empty grid to look at.

Let's add some code to **DocumentDBRepository** and **ItemController** to persist the record in DocumentDB.

1. Add the following method to your **DocumentDBRepository** class.

```
public static async Task<Document> CreateItemAsync(T item)
{
    return await client.CreateDocumentAsync(UriFactory.CreateDocumentCollectionUri(DatabaseId,
    CollectionId), item);
}
```

This method simply takes an object passed to it and persists it in DocumentDB.

2. Open the **ItemController.cs** file and add the following code snippet within the class. This is how ASP.NET MVC knows what to do for the **Create** action. In this case just render the associated **Create.cshtml** view created earlier.

```
[ActionName("Create")]
public async Task<ActionResult> CreateAsync()
{
    return View();
}
```

We now need some more code in this controller that will accept the submission from the **Create** view.

3. Add the next block of code to the `ItemController.cs` class that tells ASP.NET MVC what to do with a form POST for this controller.

```
[HttpPost]
[ActionName("Create")]
[ValidateAntiForgeryToken]
public async Task<ActionResult> CreateAsync([Bind(Include = "Id,Name,Description,Completed")] Item item)
{
    if (ModelState.IsValid)
    {
        await DocumentDBRepository<Item>.CreateItemAsync(item);
        return RedirectToAction("Index");
    }

    return View(item);
}
```

This code calls in to the `DocumentDBRepository` and uses the `CreateItemAsync` method to persist the new todo item to the database.

Security Note: The `ValidateAntiForgeryToken` attribute is used here to help protect this application against cross-site request forgery attacks. There is more to it than just adding this attribute, your views need to work with this anti-forgery token as well. For more on the subject, and examples of how to implement this correctly, please see [Preventing Cross-Site Request Forgery](#). The source code provided on [GitHub](#) has the full implementation in place.

Security Note: We also use the `Bind` attribute on the method parameter to help protect against over-posting attacks. For more details please see [Basic CRUD Operations in ASP.NET MVC](#).

This concludes the code required to add new Items to our database.

Editing Items

There is one last thing for us to do, and that is to add the ability to edit **Items** in the database and to mark them as complete. The view for editing was already added to the project, so we just need to add some code to our controller and to the `DocumentDBRepository` class again.

1. Add the following to the `DocumentDBRepository` class.

```

public static async Task<Document> UpdateItemAsync(string id, T item)
{
    return await client.ReplaceDocumentAsync(UriFactory.CreateDocumentUri(DatabaseId, CollectionId, id),
item);
}

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 == HttpStatusCode.NotFound)
        {
            return null;
        }
        else
        {
            throw;
        }
    }
}
}

```

The first of these methods, **GetItem** fetches an Item from DocumentDB which is passed back to the **ItemController** and then on to the **Edit** view.

The second of the methods we just added replaces the **Document** in DocumentDB with the version of the **Document** passed in from the **ItemController**.

2. Add the following to the **ItemController** class.

```

[HttpPost]
[ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<ActionResult> EditAsync([Bind(Include = "Id,Name,Description,Completed")] Item item)
{
    if (ModelState.IsValid)
    {
        await DocumentDBRepository<Item>.UpdateItemAsync(item.Id, item);
        return RedirectToAction("Index");
    }

    return View(item);
}

[ActionName("Edit")]
public async Task<ActionResult> EditAsync(string id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    Item item = await DocumentDBRepository<Item>.GetItemAsync(id);
    if (item == null)
    {
        return HttpNotFound();
    }

    return View(item);
}

```

The first method handles the Http GET that happens when the user clicks on the **Edit** link from the **Index** view. This method fetches a **Document** from DocumentDB and passes it to the **Edit** view.

The **Edit** view will then do an Http POST to the **IndexController**.

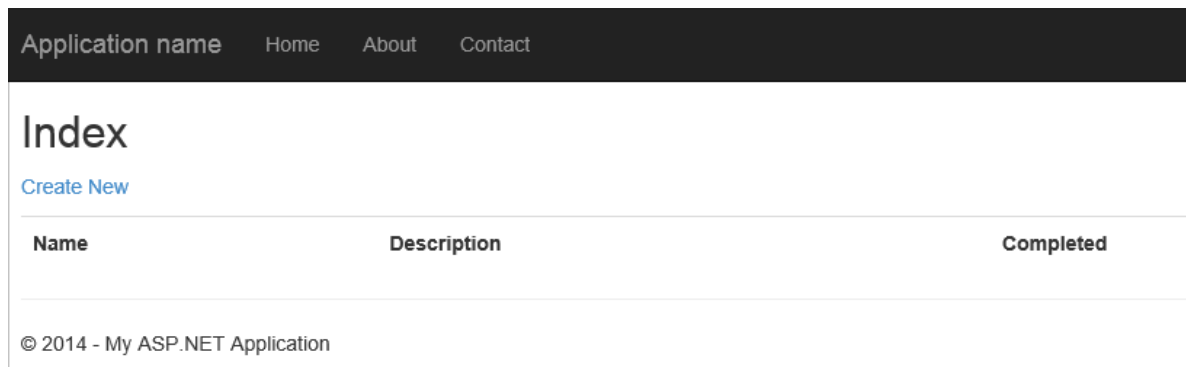
The second method we added handles passing the updated object to DocumentDB to be persisted in the database.

That's it, that is everything we need to run our application, list incomplete **Items**, add new **Items**, and edit **Items**.

Step 6: Run the application locally

To test the application on your local machine, do the following:

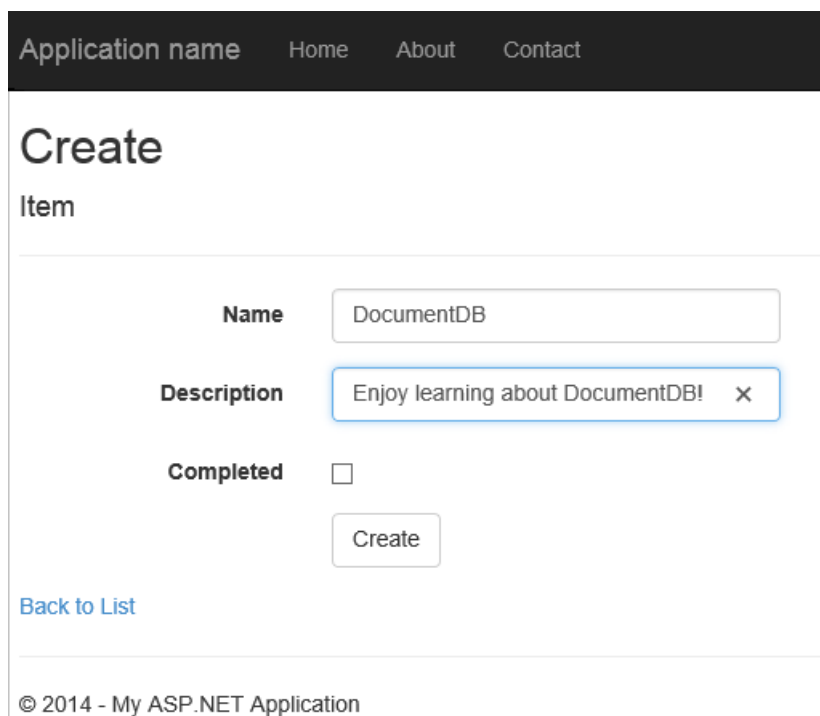
1. Hit F5 in Visual Studio to build the application in debug mode. It should build the application and launch a browser with the empty grid page we saw before:



The screenshot shows the 'Index' view of a web application. At the top is a dark navigation bar with 'Application name' and links for 'Home', 'About', and 'Contact'. Below the navigation bar, the word 'Index' is displayed in a large font. Underneath 'Index' is a link labeled 'Create New'. A table with three columns is shown: 'Name', 'Description', and 'Completed'. The table is currently empty. At the bottom of the page, there is a copyright notice: '© 2014 - My ASP.NET Application'.

If you are using Visual Studio 2013 and receive the error "Cannot await in the body of a catch clause." you need to install the [Microsoft.Net.Compilers nuget package](#). You can also compare your code against the sample project on [GitHub](#).

2. Click the **Create New** link and add values to the **Name** and **Description** fields. Leave the **Completed** check box unselected otherwise the new **Item** will be added in a completed state and will not appear on the initial list.



The screenshot shows the 'Create Item' view of the web application. It has the same dark navigation bar as the Index view. The main heading is 'Create Item'. Below this, there are three form fields: 'Name' with the value 'DocumentDB', 'Description' with the value 'Enjoy learning about DocumentDB!' (which is highlighted with a blue border), and 'Completed' with an unchecked checkbox. A 'Create' button is located below the 'Completed' checkbox. At the bottom left, there is a link labeled 'Back to List'. At the bottom of the page, there is a copyright notice: '© 2014 - My ASP.NET Application'.

3. Click **Create** and you are redirected back to the **Index** view and your **Item** appears in the list.

Application name Home About Contact		
<h2>Index</h2> Create New		
Name	Description	Completed
DocumentDB	Enjoy learning about DocumentDB!	<input type="checkbox"/>
© 2014 - My ASP.NET Application		

Feel free to add a few more **Items** to your todo list.

- Click **Edit** next to an **Item** on the list and you are taken to the **Edit** view where you can update any property of your object, including the **Completed** flag. If you mark the **Complete** flag and click **Save**, the **Item** is removed from the list of incomplete tasks.

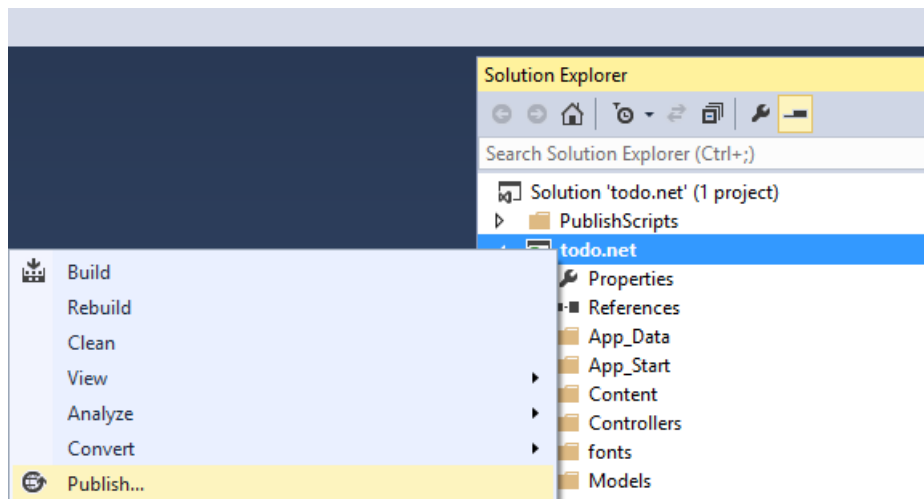
Application name Home About Contact	
<h2>Edit</h2> <p>Item</p>	
Name	<input type="text" value="DocumentDB"/>
Description	<input type="text" value="Enjoy learning about DocumentDB!"/>
Completed	<input checked="" type="checkbox"/>
	<input type="button" value="Save"/>
Back to List	

- Once you've tested the app, press Ctrl+F5 to stop debugging the app. You're ready to deploy!

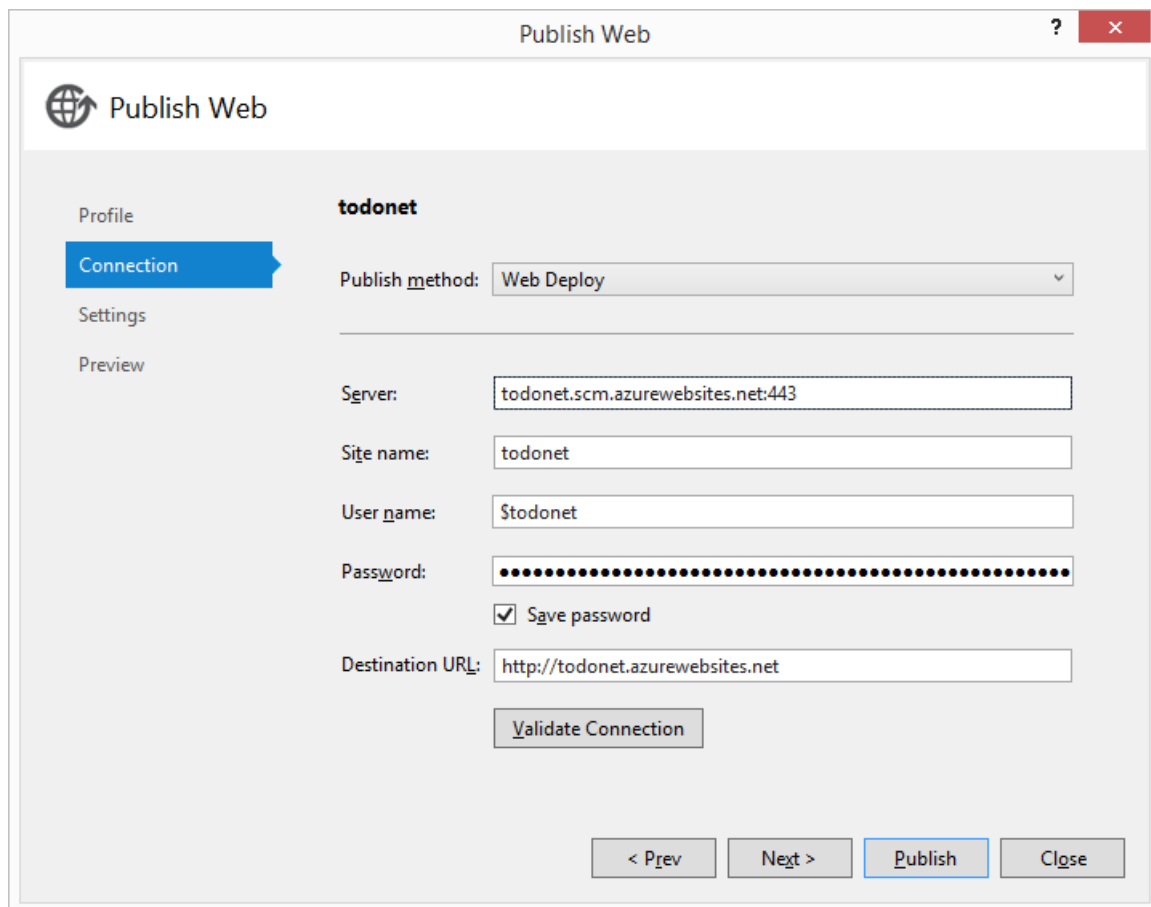
Step 7: Deploy the application to Azure Websites

Now that you have the complete application working correctly with DocumentDB we're going to deploy this web app to Azure Websites. If you selected **Host in the cloud** when you created the empty ASP.NET MVC project then Visual Studio makes this really easy and does most of the work for you.

- To publish this application all you need to do is right-click on the project in **Solution Explorer** and click **Publish**.



2. Everything should already be configured according to your credentials; in fact the website has already been created in Azure for you at the **Destination URL** shown, all you need to do is click **Publish**.



In a few seconds, Visual Studio will finish publishing your web application and launch a browser where you can see your handy work running in Azure!

Troubleshooting

If you receive the "An error occurred while processing your request" while trying to deploy the web app, do the following:

1. Cancel out of the error message and then select **Microsoft Azure Web Apps** again.
2. Login and then select **New** to create a new web app.
3. On the **Create a Web App on Microsoft Azure** screen, do the following:
 - Web App name: "todo-net-app"

- App Service plan: Create new, named "todo-net-app"
 - Resource group: Create new, named "todo-net-app"
 - Region: Select the region closest to your app users
 - Database server: Click no database, then click **Create**.
4. In the "todo-net-app * screen", click **Validate Connection**. After the connection is verified, **Publish**.

The app then gets displayed on your browser.

Next steps

Congratulations! You just built your first ASP.NET MVC web application using Azure DocumentDB and published it to Azure Websites. The source code for the complete application, including the detail and delete functionality that were not included in this tutorial can be downloaded or cloned from [GitHub](#). So if you're interested in adding that to your app, grab the code and add it to this app.

To add additional functionality to your application, review the APIs available in the [DocumentDB .NET Library](#) and feel free to contribute to the DocumentDB .NET Library on [GitHub](#).

Build a Node.js web application using DocumentDB

11/22/2016 • 13 min to read • [Edit on GitHub](#)

Contributors

[Syam Nair](#) • [mimig](#) • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [carolinacmoravia](#) • [Andrew Hoh](#) • [Andrew Liu](#) • [Cephas Lin](#) • [arramac](#) • [Ryan CrawCour](#) • [v-aljenk](#) • [Tom Dykstra](#) • [Dene Hager](#)

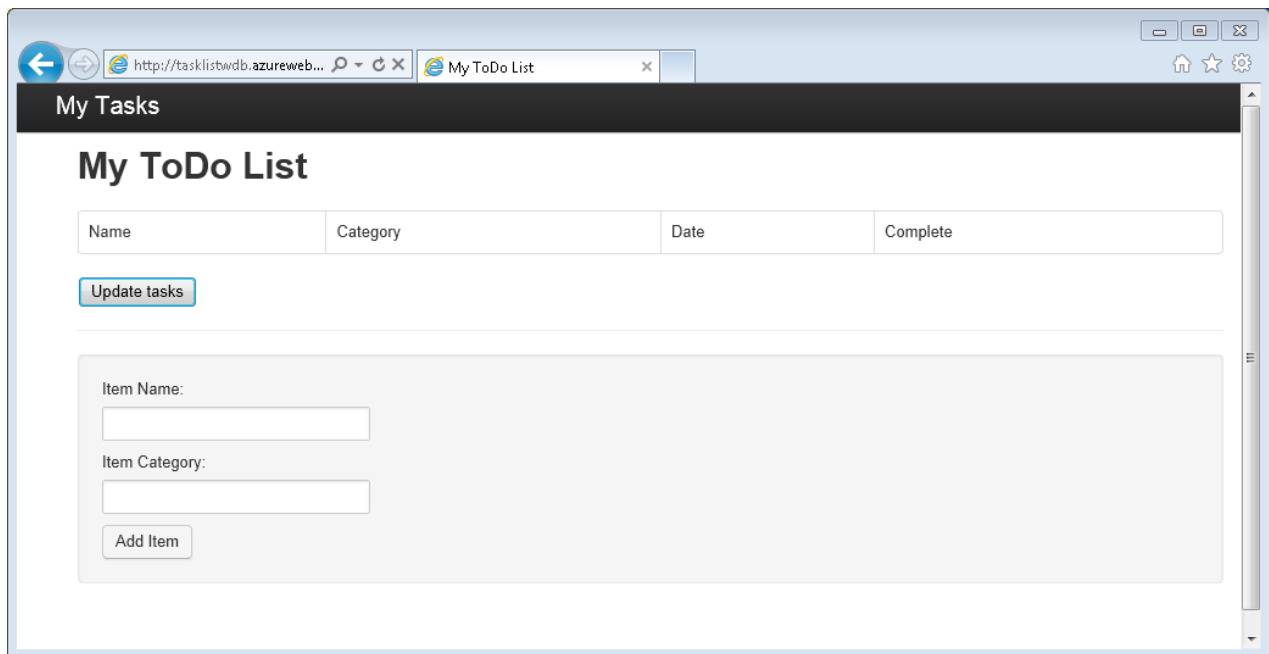
This Node.js tutorial shows you how to use the Azure DocumentDB service to store and access data from a Node.js Express application hosted on Azure Websites.

We recommend getting started by watching the following video, where you will learn how to provision an Azure DocumentDB database account and store JSON documents in your Node.js application.

Then, return to this Node.js tutorial, where you'll learn the answers to the following questions:

- How do I work with DocumentDB using the documentdb npm module?
- How do I deploy the web application to Azure Websites?

By following this database tutorial, you will build a simple web-based task-management application that allows creating, retrieving and completing of tasks. The tasks will be stored as JSON documents in Azure DocumentDB.



Don't have time to complete the tutorial and just want to get the complete solution? Not a problem, you can get the complete sample solution from [GitHub](#).

Prerequisites

TIP

This Node.js tutorial assumes that you have some prior experience using Node.js and Azure Websites.

Before following the instructions in this article, you should ensure that you have the following:

- An active Azure account. If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see [Azure Free Trial](#)

OR

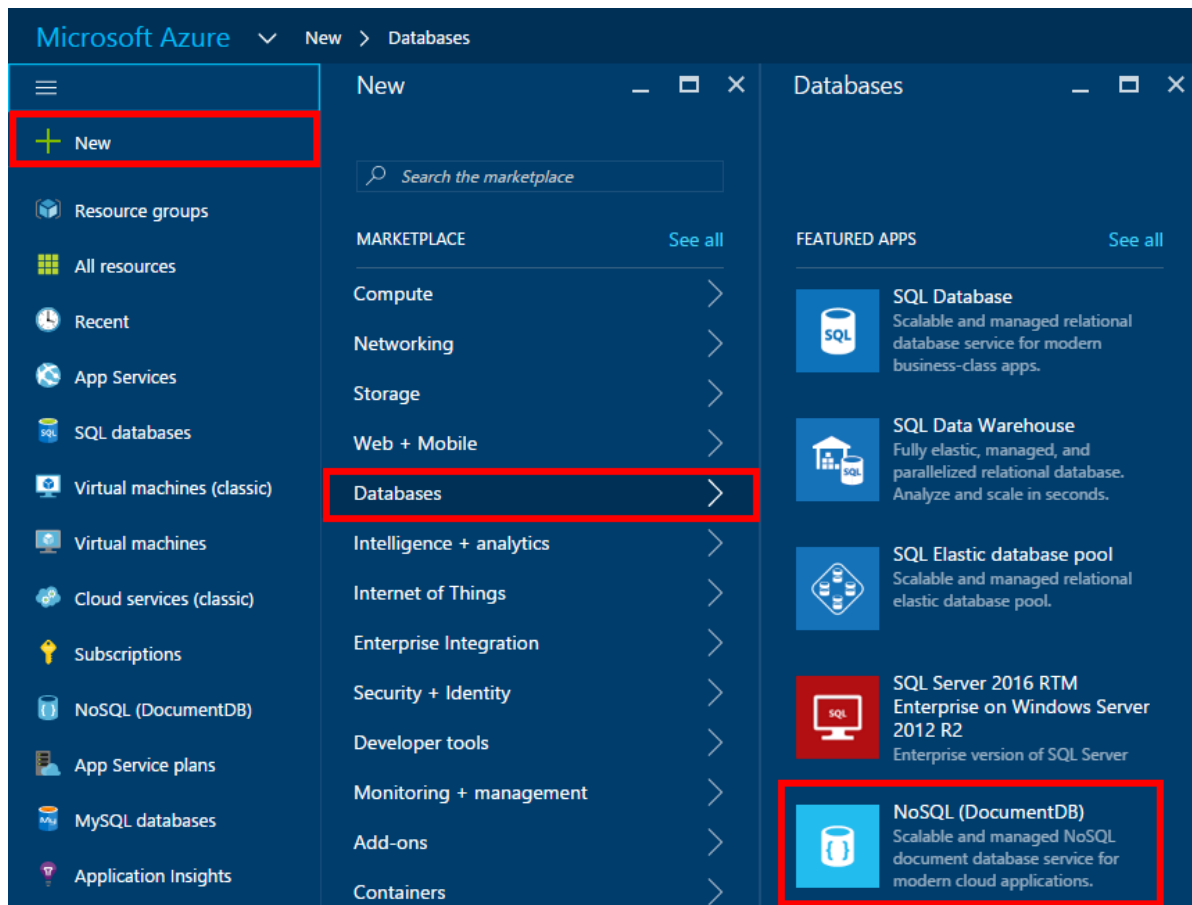
A local installation of the [Azure DocumentDB Emulator](#).

- [Node.js](#) version v0.10.29 or higher.
- [Express generator](#) (you can install this via `npm install express-generator -g`)
- [Git](#).

Step 1: Create a DocumentDB database account

Let's start by creating a DocumentDB account. If you already have an account or if you are using the DocumentDB Emulator for this tutorial, you can skip to [Step 2: Create a new Node.js application](#).

1. Sign in to the [Azure portal](#).
2. In the Jumpbar, click **New**, click **Databases**, and then click **NoSQL (DocumentDB)**.



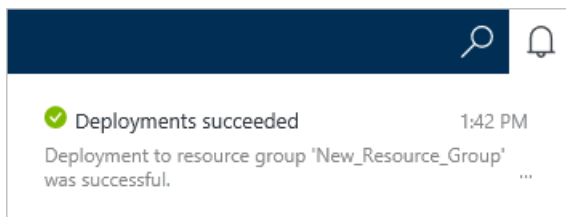
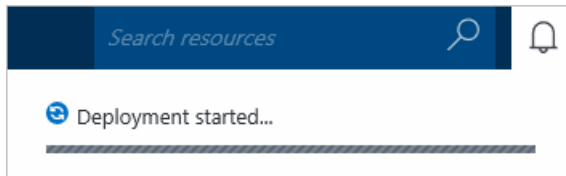
3. In the **New** account blade, specify the desired configuration for the DocumentDB account.

The screenshot shows the 'NoSQL (DocumentDB) New account' configuration form. The form includes the following fields and options:

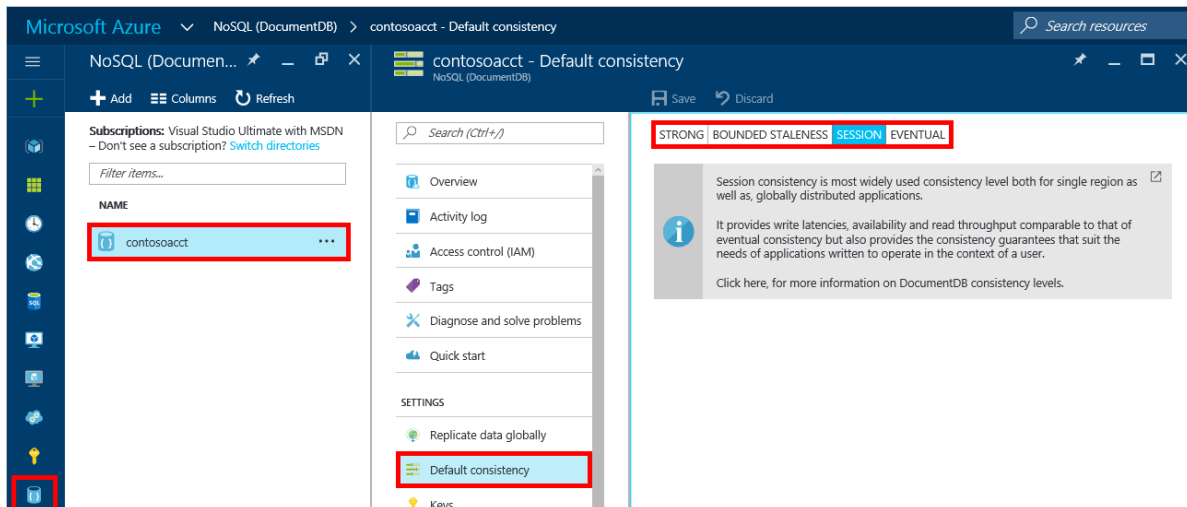
- ID:** A text box containing 'contosoacct' with a green checkmark indicating validation. Below it, the text 'documents.azure.com' is displayed.
- NoSQL API:** Two buttons: 'DocumentDB' (selected) and 'MongoDB'.
- Subscription:** A dropdown menu showing 'Visual Studio Ultimate with MSDN'.
- Resource Group:** Radio buttons for 'Create new' (selected) and 'Use existing'. Below, a text box contains 'contosoacct' with a green checkmark.
- Location:** A dropdown menu showing 'West US'.
- Pin to dashboard:** An unchecked checkbox.
- Buttons:** A blue 'Create' button and a blue link for 'Automation options'.

- In the **ID** box, enter a name to identify the DocumentDB account. When the **ID** is validated, a green check mark appears in the **ID** box. The **ID** value becomes the host name within the URI. The **ID** may contain only lowercase letters, numbers, and the '-' character, and must be between 3 and 50 characters. Note that *documents.azure.com* is appended to the endpoint name you choose, the result of which becomes your DocumentDB account endpoint.

- In the **NoSQL API** box, select **DocumentDB**.
 - For **Subscription**, select the Azure subscription that you want to use for the DocumentDB account. If your account has only one subscription, that account is selected by default.
 - In **Resource Group**, select or create a resource group for your DocumentDB account. By default, a new resource group is created. For more information, see [Using the Azure portal to manage your Azure resources](#).
 - Use **Location** to specify the geographic location in which to host your DocumentDB account.
4. Once the new DocumentDB account options are configured, click **Create**. To check the status of the deployment, check the Notifications hub.



5. After the DocumentDB account is created, it is ready for use with the default settings. To review the default settings, click the **NoSQL (DocumentDB)** icon on the Jumpbar, click your new account, and then click **Default Consistency** in the Resource Menu.



The default consistency of the DocumentDB account is set to **Session**. You can adjust the default consistency by clicking **Default Consistency** in the resource menu. To learn more about the consistency levels offered by DocumentDB, see [Consistency levels in DocumentDB](#).

Now navigate to the DocumentDB account blade, and click **Keys**, as we will use these values in the web application we create next.

contoso1 - Keys

contoso1

Search (Ctrl+*/*)

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Tutorials and resources

SETTINGS

Replicate data globally

Default consistency

Keys

Read-write Keys Read-only Keys

URI

https://contoso1.documents.azure.com:443/

PRIMARY KEY

4jN1mqnm7050oLUtz7CiscskeE9sdbMvaGUsB1KKWknoLRSDStV75u0HGUxEGLH0PgXITV

SECONDARY KEY

nJmxlukeNDReMI2ZJK0XzXXOiPqD4m6EI3fbsKe2dKofMilwgbhrOBgA736bUsb8Pydvobyl

PRIMARY CONNECTION STRING

AccountEndpoint=https://contoso1.documents.azure.com:443/;AccountKey=4jN1mqnm7

SECONDARY CONNECTION STRING

AccountEndpoint=https://contoso1.documents.azure.com:443/;AccountKey=nJmxlukeNC

Step 2: Learn to create a new Node.js application

Now let's learn to create a basic Hello World Node.js project using the [Express](#) framework.

1. Open your favorite terminal.
2. Use the express generator to generate a new application called **todo**.

```
express todo
```

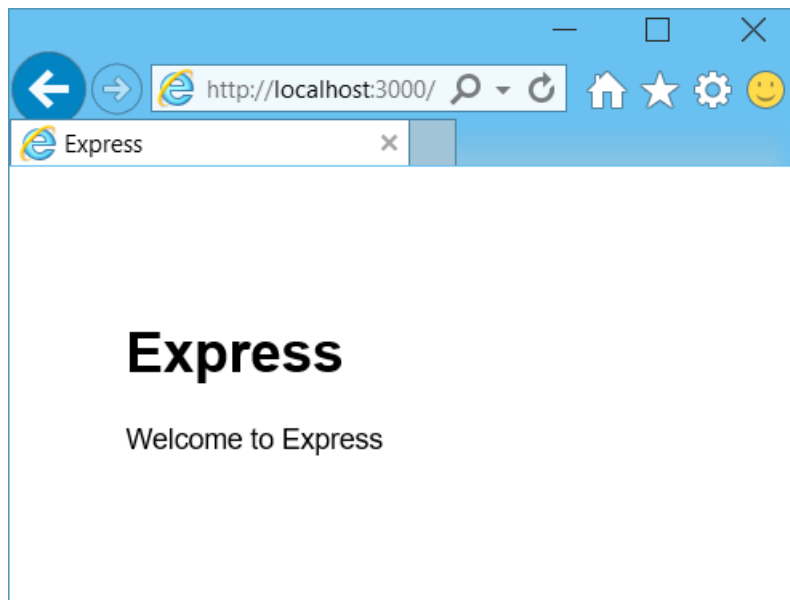
3. Open your new **todo** directory and install dependencies.

```
cd todo  
npm install
```

4. Run your new application.

```
npm start
```

5. You can view your new application by navigating your browser to <http://localhost:3000>.



Step 3: Install additional modules

The **package.json** file is one of the files created in the root of the project. This file contains a list of additional modules that are required for your Node.js application. Later, when you deploy this application to an Azure Websites, this file is used to determine which modules need to be installed on Azure to support your application. We still need to install two more packages for this tutorial.

1. Back in the terminal, install the **async** module via npm.

```
npm install async --save
```

2. Install the **documentdb** module via npm. This is the module where all the DocumentDB magic happens.

```
npm install documentdb --save
```

3. A quick check of the **package.json** file of the application should show the additional modules. This file will tell Azure which packages to download and install when running your application. It should resemble the example below.

```

1  {
2    "name": "Todo",
3    "version": "0.0.0",
4    "private": true,
5    "scripts": {
6      "start": "node ./bin/www"
7    },
8    "description": "Building a Todo List application with DocumentDB",
9    "author": {
10     "name": "",
11     "email": ""
12   },
13
14   "dependencies": {
15     "async": "^0.9.0",
16     "body-parser": "~1.10.2",
17     "cookie-parser": "~1.3.3",
18     "debug": "~2.1.1",
19     "documentdb": "^1.0.0",
20     "express": "~4.11.1",
21     "jade": "~1.9.1",
22     "morgan": "~1.5.1",
23     "serve-favicon": "~2.2.0"
24   }
25 }
26

```

This tells Node (and Azure later) that your application depends on these additional modules.

Step 4: Using the DocumentDB service in a node application

That takes care of all the initial setup and configuration, now let's get down to why we're here, and that's to write some code using Azure DocumentDB.

Create the model

1. In the project directory, create a new directory named **models**.
2. In the **models** directory, create a new file named **taskDao.js**. This file will contain the model for the tasks created by our application.
3. In the same **models** directory, create another new file named **docdbUtils.js**. This file will contain some useful, reusable, code that we will use throughout our application.
4. Copy the following code in to **docdbUtils.js**

```

var DocumentDBClient = require('documentdb').DocumentClient;

var DocDBUtils = {
  getOrCreateDatabase: function (client, databaseId, callback) {
    var querySpec = {
      query: 'SELECT * FROM root r WHERE r.id= @id',
      parameters: [{
        name: '@id',
        value: databaseId
      }]
    };

    client.queryDatabases(querySpec).toArray(function (err, results) {
      if (err) {
        callback(err);

      } else {
        if (results.length === 0) {
          var databaseSpec = {
            id: databaseId
          };

          client.createDatabase(databaseSpec, function (err, created) {
            callback(null, created);
          });

        } else {
          callback(null, results[0]);
        }
      }
    });
  },

  getOrCreateCollection: function (client, databaseLink, collectionId, callback) {
    var querySpec = {
      query: 'SELECT * FROM root r WHERE r.id=@id',
      parameters: [{
        name: '@id',
        value: collectionId
      }]
    };

    client.queryCollections(databaseLink, querySpec).toArray(function (err, results) {
      if (err) {
        callback(err);

      } else {
        if (results.length === 0) {
          var collectionSpec = {
            id: collectionId
          };

          client.createCollection(databaseLink, collectionSpec, function (err, created) {
            callback(null, created);
          });

        } else {
          callback(null, results[0]);
        }
      }
    });
  }
};

module.exports = DocDBUtils;

```

TIP

`createCollection` takes an optional `requestOptions` parameter that can be used to specify the Offer Type for the Collection. If no `requestOptions.offerType` value is supplied then the Collection will be created using the default Offer Type.

For more information on DocumentDB Offer Types please refer to [Performance levels in DocumentDB](#)

5. Save and close the `docdbUtils.js` file.
6. At the beginning of the `taskDao.js` file, add the following code to reference the `DocumentDBClient` and the `docdbUtils.js` we created above:

```
var DocumentDBClient = require('documentdb').DocumentClient;
var docdbUtils = require('./docdbUtils');
```

7. Next, you will add code to define and export the Task object. This is responsible for initializing our Task object and setting up the Database and Document Collection we will use.

```
function TaskDao(documentDBClient, databaseId, collectionId) {
  this.client = documentDBClient;
  this.databaseId = databaseId;
  this.collectionId = collectionId;

  this.database = null;
  this.collection = null;
}

module.exports = TaskDao;
```

8. Next, add the following code to define additional methods on the Task object, which allow interactions with data stored in DocumentDB.

```
TaskDao.prototype = {
  init: function (callback) {
    var self = this;

    docdbUtils.getOrCreateDatabase(self.client, self.databaseId, function (err, db) {
      if (err) {
        callback(err);
      } else {
        self.database = db;
        docdbUtils.getOrCreateCollection(self.client, self.database._self, self.collectionId,
function (err, coll) {
          if (err) {
            callback(err);

          } else {
            self.collection = coll;
          }
        });
      }
    });
  },

  find: function (querySpec, callback) {
    var self = this;

    self.client.queryDocuments(self.collection._self, querySpec).toArray(function (err, results) {
      if (err) {
        callback(err);

      } else {
```

```

        callback(null, results);
    }
    });
},

addItem: function (item, callback) {
    var self = this;

    item.date = Date.now();
    item.completed = false;

    self.client.createDocument(self.collection._self, item, function (err, doc) {
        if (err) {
            callback(err);

        } else {
            callback(null, doc);
        }
    });
},

updateItem: function (itemId, callback) {
    var self = this;

    self.getItem(itemId, function (err, doc) {
        if (err) {
            callback(err);

        } else {
            doc.completed = true;

            self.client.replaceDocument(doc._self, doc, function (err, replaced) {
                if (err) {
                    callback(err);

                } else {
                    callback(null, replaced);
                }
            });
        }
    });
},

getItem: function (itemId, callback) {
    var self = this;

    var querySpec = {
        query: 'SELECT * FROM root r WHERE r.id = @id',
        parameters: [{
            name: '@id',
            value: itemId
        }]
    };

    self.client.queryDocuments(self.collection._self, querySpec).toArray(function (err, results) {
        if (err) {
            callback(err);

        } else {
            callback(null, results[0]);
        }
    });
}
};

```

9. Save and close the **taskDao.js** file.

Create the controller

1. In the **routes** directory of your project, create a new file named **tasklist.js**.
2. Add the following code to **tasklist.js**. This loads the DocumentDBClient and async modules, which are used by **tasklist.js**. This also defined the **TaskList** function, which is passed an instance of the **Task** object we defined earlier:

```
var DocumentDBClient = require('documentdb').DocumentClient;
var async = require('async');

function TaskList(taskDao) {
  this.taskDao = taskDao;
}

module.exports = TaskList;
```

3. Continue adding to the **tasklist.js** file by adding the methods used to **showTasks**, **addTask**, and **completeTasks**:

```

TaskList.prototype = {
  showTasks: function (req, res) {
    var self = this;

    var querySpec = {
      query: 'SELECT * FROM root r WHERE r.completed=@completed',
      parameters: [{
        name: '@completed',
        value: false
      }]
    };

    self.taskDao.find(querySpec, function (err, items) {
      if (err) {
        throw (err);
      }

      res.render('index', {
        title: 'My ToDo List ',
        tasks: items
      });
    });
  },

  addTask: function (req, res) {
    var self = this;
    var item = req.body;

    self.taskDao.addItem(item, function (err) {
      if (err) {
        throw (err);
      }

      res.redirect('/');
    });
  },

  completeTask: function (req, res) {
    var self = this;
    var completedTasks = Object.keys(req.body);

    async.forEach(completedTasks, function taskIterator(completedTask, callback) {
      self.taskDao.updateItem(completedTask, function (err) {
        if (err) {
          callback(err);
        } else {
          callback(null);
        }
      });
    }, function goHome(err) {
      if (err) {
        throw err;
      } else {
        res.redirect('/');
      }
    });
  }
};

```

4. Save and close the **tasklist.js** file.

Add config.js

1. In your project directory create a new file named **config.js**.
2. Add the following to **config.js**. This defines configuration settings and values needed for our application.

```

var config = {}

config.host = process.env.HOST || "[the URI value from the DocumentDB Keys blade on
http://portal.azure.com]";
config.authKey = process.env.AUTH_KEY || "[the PRIMARY KEY value from the DocumentDB Keys blade on
http://portal.azure.com]";
config.databaseId = "ToDoList";
config.collectionId = "Items";

module.exports = config;

```

3. In the **config.js** file, update the values of HOST and AUTH_KEY using the values found in the Keys blade of your DocumentDB account on the [Microsoft Azure Portal](#):
4. Save and close the **config.js** file.

Modify app.js

1. In the project directory, open the **app.js** file. This file was created earlier when the Express web application was created.
2. Add the following code to the top of **app.js**

```

var DocumentDBClient = require('documentdb').DocumentClient;
var config = require('./config');
var TaskList = require('./routes/tasklist');
var TaskDao = require('./models/taskDao');

```

3. This code defines the config file to be used, and proceeds to read values out of this file in to some variables we will use soon.
4. Replace the following two lines in **app.js** file:

```

app.use('/', routes);
app.use('/users', users);

```

with the following snippet:

```

var docDbClient = new DocumentDBClient(config.host, {
  masterKey: config.authKey
});
var taskDao = new TaskDao(docDbClient, config.databaseId, config.collectionId);
var taskList = new TaskList(taskDao);
taskDao.init();

app.get('/', taskList.showTasks.bind(taskList));
app.post('/addtask', taskList.addTask.bind(taskList));
app.post('/completetask', taskList.completeTask.bind(taskList));
app.set('view engine', 'jade');

```

5. These lines define a new instance of our **TaskDao** object, with a new connection to DocumentDB (using the values read from the **config.js**), initialize the task object and then bind form actions to methods on our **TaskList** controller.
6. Finally, save and close the **app.js** file, we're just about done.

Step 5: Build a user interface

Now let's turn our attention to building the user interface so a user can actually interact with our application. The Express application we created uses **Jade** as the view engine. For more information on Jade please refer to <http://jade-lang.com/>.

1. The **layout.jade** file in the **views** directory is used as a global template for other **.jade** files. In this step you will modify it to use **Twitter Bootstrap**, which is a toolkit that makes it easy to design a nice looking website.
2. Open the **layout.jade** file found in the **views** folder and replace the contents with the following;

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='//ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/css/bootstrap.min.css')
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    nav.navbar.navbar-inverse.navbar-fixed-top
      div.navbar-header
        a.navbar-brand(href='#') My Tasks
      block content
    script(src='//ajax.aspnetcdn.com/ajax/jquery/jquery-1.11.2.min.js')
    script(src='//ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/bootstrap.min.js')
```

This effectively tells the **Jade** engine to render some HTML for our application and creates a **block** called **content** where we can supply the layout for our content pages. Save and close this **layout.jade** file.

3. Now open the **index.jade** file, the view that will be used by our application, and replace the content of the file with the following:

```
extends layout

block content
  h1 #{title}
  br

  form(action="/completetask", method="post")
    table.table.table-striped.table-bordered
      tr
        td Name
        td Category
        td Date
        td Complete
      if (typeof tasks === "undefined")
        tr
          td
      else
        each task in tasks
          tr
            td #{task.name}
            td #{task.category}
            - var date = new Date(task.date);
            - var day = date.getDate();
            - var month = date.getMonth() + 1;
            - var year = date.getFullYear();
            td #{month + "/" + day + "/" + year}
            td
              input(type="checkbox", name="#{task.id}", value="#{!task.completed}",
checked=task.completed)
            button.btn(type="submit") Update tasks
    hr
  form.well(action="/addtask", method="post")
    label Item Name:
    input(name="name", type="text")
    label Item Category:
    input(name="category", type="text")
    br
    button.btn(type="submit") Add item
```

This extends layout, and provides content for the **content** placeholder we saw in the **layout.jade** file

earlier.

In this layout we created two HTML forms. The first form contains a table for our data and a button that allows us to update items by posting to `/completetask` method of our controller. The second form contains two input fields and a button that allows us to create a new item by posting to `/addtask` method of our controller.

This should be all that we need for our application to work.

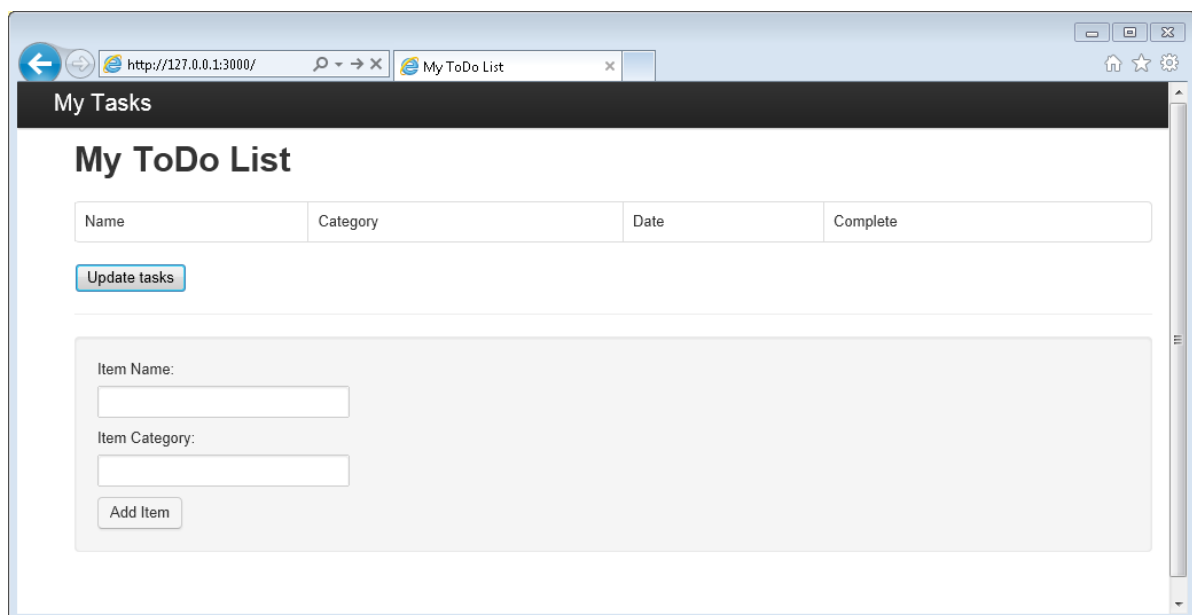
4. Open the `style.css` file in `public\stylesheets` directory and replace the code with the following:

```
body {
  padding: 50px;
  font: 14px "Lucida Grande", Helvetica, Arial, sans-serif;
}
a {
  color: #00B7FF;
}
.well label {
  display: block;
}
.well input {
  margin-bottom: 5px;
}
.btn {
  margin-top: 5px;
  border: outset 1px #C8C8C8;
}
```

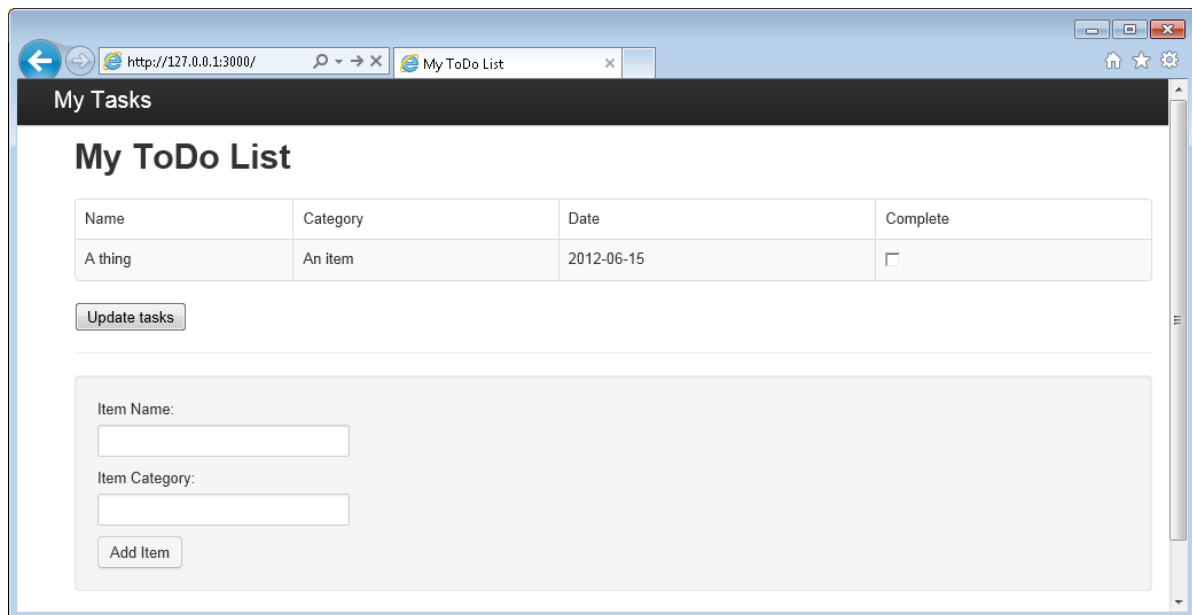
Save and close this `style.css` file.

Step 6: Run your application locally

1. To test the application on your local machine, run `npm start` in a terminal to start your application, and launch a browser with a page that looks like the image below:



2. Use the provided fields for Item, Item Name and Category to enter information, and then click **Add Item**.
3. The page should update to display the newly created item in the ToDo list.



4. To complete a task, simply check the checkbox in the Complete column, and then click **Update tasks**.

Step 7: Deploy your application development project to Azure Websites

1. If you haven't already, enable a git repository for your Azure Website. You can find instructions on how to do this in the [Local Git Deployment to Azure App Service](#) topic.
2. Add your Azure Website as a git remote.

```
git remote add azure https://username@your-azure-website.scm.azurewebsites.net:443/your-azure-website.git
```

3. Deploy by pushing to the remote.

```
git push azure master
```

4. In a few seconds, git will finish publishing your web application and launch a browser where you can see your handy work running in Azure!

Next steps

Congratulations! You have just built your first Node.js Express Web Application using Azure DocumentDB and published it to Azure Websites.

The source code for the complete reference application can be downloaded from [GitHub](#).

For more information, see the [Node.js Developer Center](#).

Build a Java web application using DocumentDB

11/22/2016 • 18 min to read • [Edit on GitHub](#)

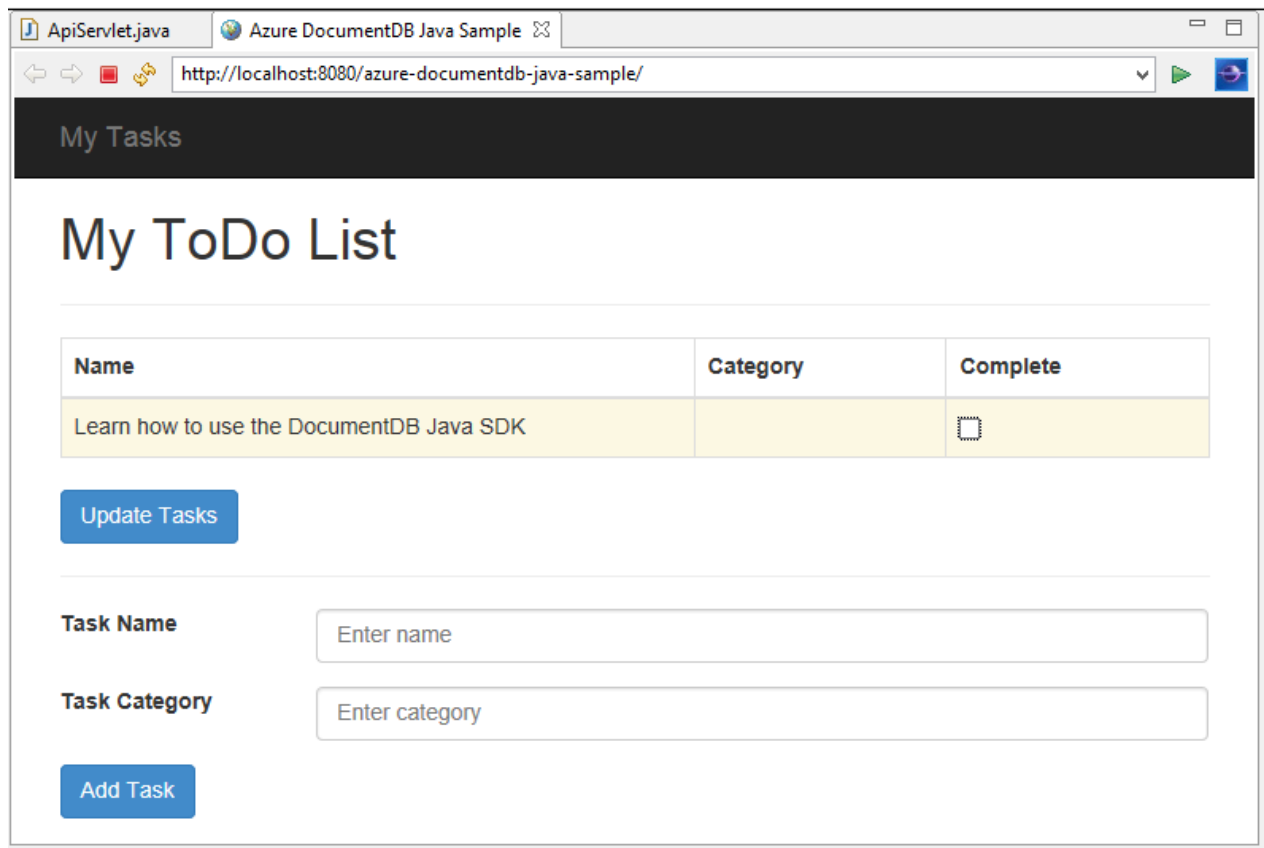
Contributors

Denny Lee • mimig • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Andrew Liu • Andrew Hoh • v-aljenk
• Dene Hager

This Java web application tutorial shows you how to use the [Microsoft Azure DocumentDB](#) service to store and access data from a Java application hosted on Azure Websites. In this topic, you will learn:

- How to build a basic JSP application in Eclipse.
- How to work with the Azure DocumentDB service using the [DocumentDB Java SDK](#).

This Java application tutorial shows you how to create a web-based task-management application that enables you to create, retrieve, and mark tasks as complete, as shown in the following image. Each of the tasks in the ToDo list are stored as JSON documents in Azure DocumentDB.



The screenshot shows a web browser window with the URL `http://localhost:8080/azure-documentdb-java-sample/`. The page has a dark header with the text "My Tasks". Below the header, the main content area is titled "My ToDo List". It contains a table with three columns: "Name", "Category", and "Complete". The table has one row with the text "Learn how to use the DocumentDB Java SDK" in the "Name" column and a checkbox in the "Complete" column. Below the table is a blue button labeled "Update Tasks". At the bottom of the page, there are two input fields: "Task Name" with the placeholder text "Enter name" and "Task Category" with the placeholder text "Enter category". Below these fields is a blue button labeled "Add Task".

Name	Category	Complete
Learn how to use the DocumentDB Java SDK		<input type="checkbox"/>

Update Tasks

Task Name

Task Category

Add Task

TIP

This application development tutorial assumes that you have prior experience using Java. If you are new to Java or the [prerequisite tools](#), we recommend downloading the complete [todo](#) project from GitHub and building it using [the instructions at the end of this article](#). Once you have it built, you can review the article to gain insight on the code in the context of the project.

Prerequisites for this Java web application tutorial

Before you begin this application development tutorial, you must have the following:

- An active Azure account. If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see [Azure Free Trial](#)

OR

A local installation of the [Azure DocumentDB Emulator](#).

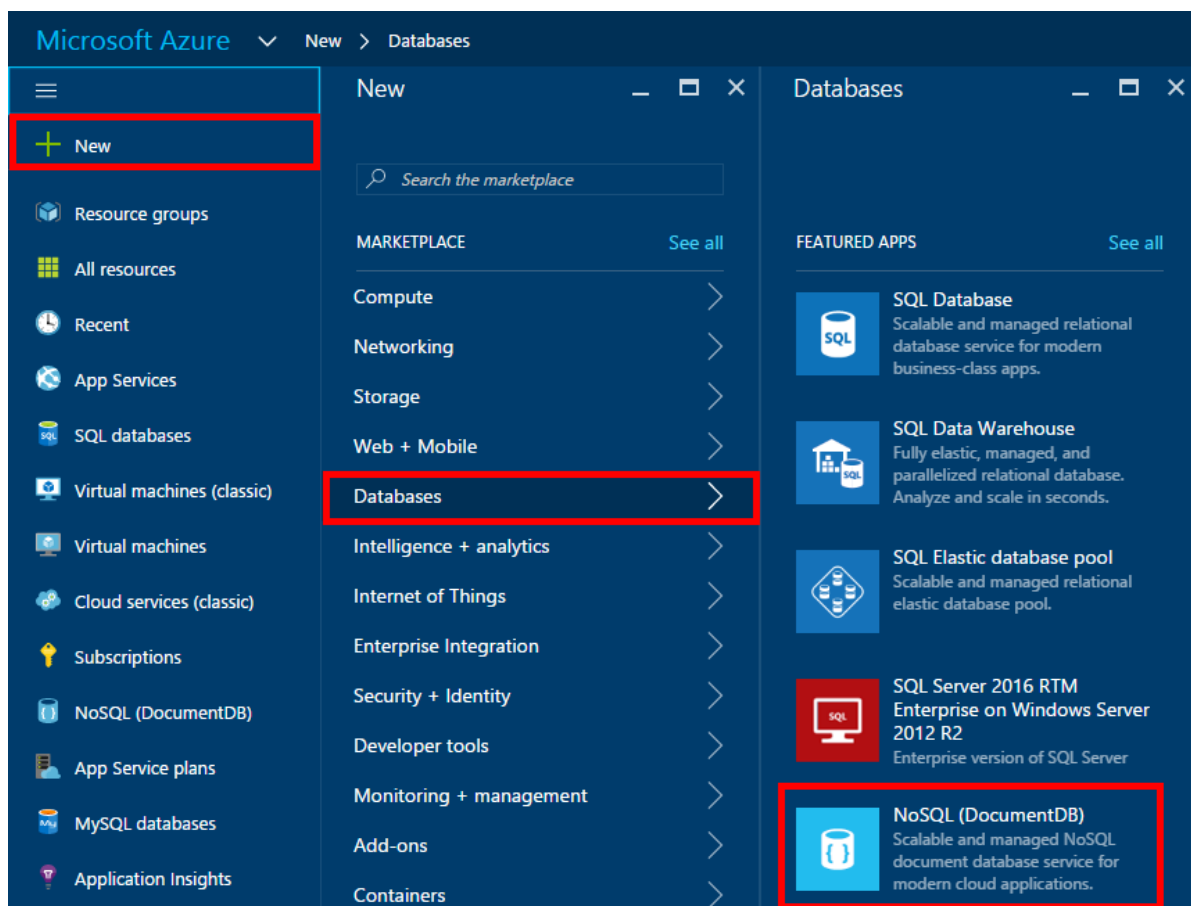
- [Java Development Kit \(JDK\) 7+](#).
- [Eclipse IDE for Java EE Developers](#).
- [An Azure Website with a Java runtime environment \(e.g. Tomcat or Jetty\) enabled](#).

If you're installing these tools for the first time, coreservlets.com provides a walk-through of the installation process in the Quick Start section of their [Tutorial: Installing TomCat7 and Using it with Eclipse](#) article.

Step 1: Create a DocumentDB database account

Let's start by creating a DocumentDB account. If you already have an account or if you are using the DocumentDB Emulator for this tutorial, you can skip to [Step 2: Create the Java JSP application](#).

1. Sign in to the [Azure portal](#).
2. In the Jumpbar, click **New**, click **Databases**, and then click **NoSQL (DocumentDB)**.



3. In the **New** account blade, specify the desired configuration for the DocumentDB account.

NoSQL (DocumentDB)
New account

* ID
contosoacct ✓
documents.azure.com

NoSQL API ⓘ
DocumentDB MongoDB

* Subscription
Visual Studio Ultimate with MSDN ▼

* Resource Group ⓘ
☒ Create new ☐ Use existing
contosoacct ✓

* Location
West US ▼

☐ Pin to dashboard

Create Automation options

- In the **ID** box, enter a name to identify the DocumentDB account. When the **ID** is validated, a green check mark appears in the **ID** box. The **ID** value becomes the host name within the URI. The **ID** may contain only lowercase letters, numbers, and the '-' character, and must be between 3 and 50 characters. Note that *documents.azure.com* is appended to the endpoint name you choose, the result of which becomes your DocumentDB account endpoint.
 - In the **NoSQL API** box, select **DocumentDB**.
 - For **Subscription**, select the Azure subscription that you want to use for the DocumentDB account. If your account has only one subscription, that account is selected by default.
 - In **Resource Group**, select or create a resource group for your DocumentDB account. By default, a new resource group is created. For more information, see [Using the Azure portal to manage your Azure resources](#).
 - Use **Location** to specify the geographic location in which to host your DocumentDB account.
4. Once the new DocumentDB account options are configured, click **Create**. To check the status of the deployment, check the Notifications hub.

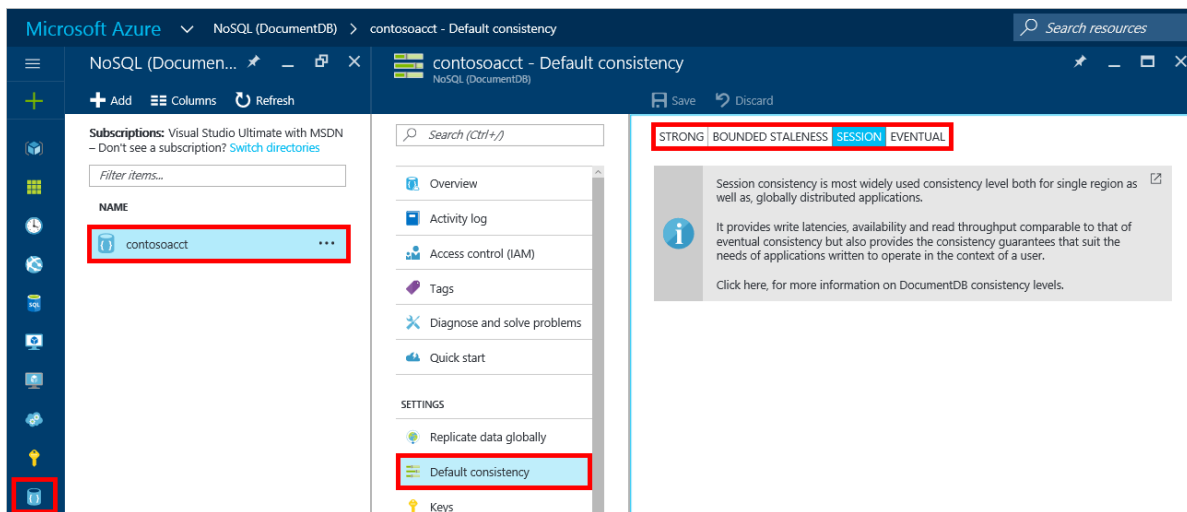
Search resources 🔍 🔔

🔄 Deployment started...

🔍 🔔

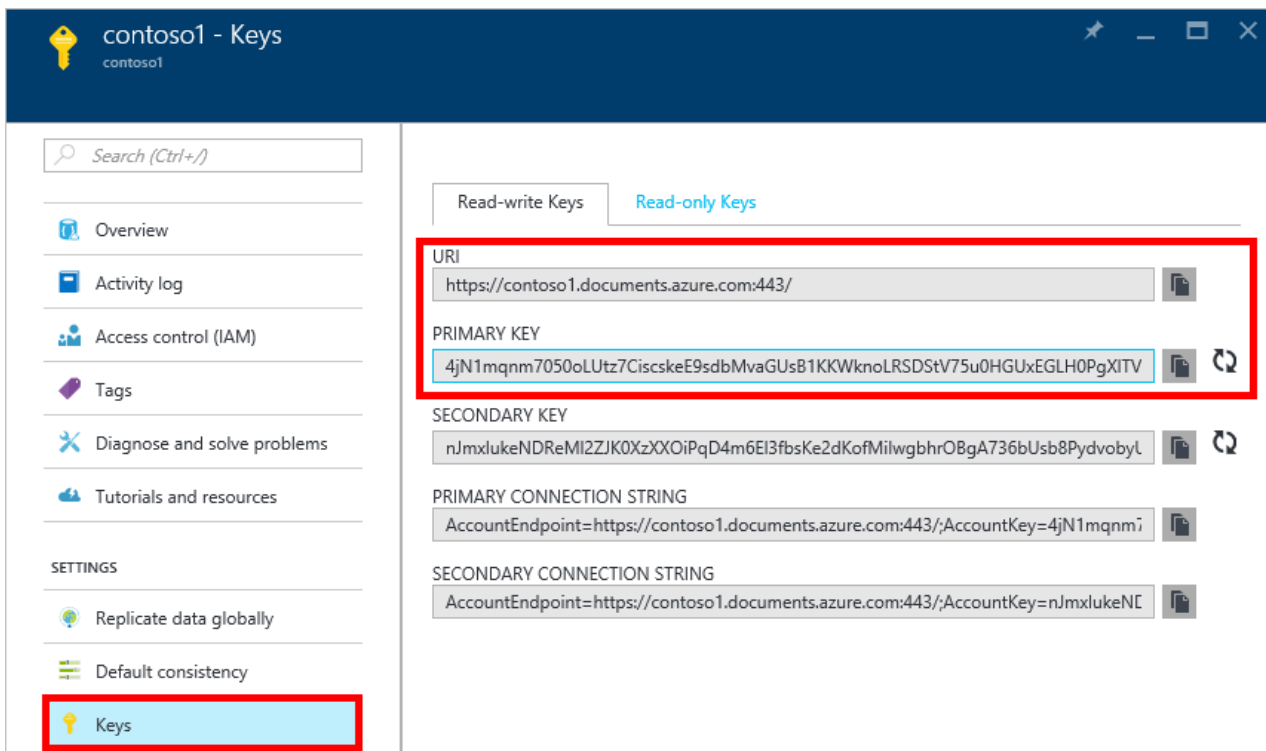
✓ Deployments succeeded 1:42 PM
Deployment to resource group 'New_Resource_Group' was successful. ...

5. After the DocumentDB account is created, it is ready for use with the default settings. To review the default settings, click the **NoSQL (DocumentDB)** icon on the Jumpbar, click your new account, and then click **Default Consistency** in the Resource Menu.



The default consistency of the DocumentDB account is set to **Session**. You can adjust the default consistency by clicking **Default Consistency** in the resource menu. To learn more about the consistency levels offered by DocumentDB, see [Consistency levels in DocumentDB](#).

Now navigate to the DocumentDB account blade, and click **Keys**, as we will use these values in the web application we create next.



Step 2: Create the Java JSP application

To create the JSP application:

1. First, we'll start off by creating a Java project. Start Eclipse, then click **File**, click **New**, and then click **Dynamic Web Project**. If you don't see **Dynamic Web Project** listed as an available project, do the following: click **File**, click **New**, click **Project...**, expand **Web**, click **Dynamic Web Project**, and click **Next**.

New Dynamic Web Project

Dynamic Web Project
Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: MyHelloWorld

Project location
☒ Use default location
Location: C:\eclipse_data\MyHelloWorld Browse...

Target runtime
<None> New Runtime...

Dynamic web module version
3.0

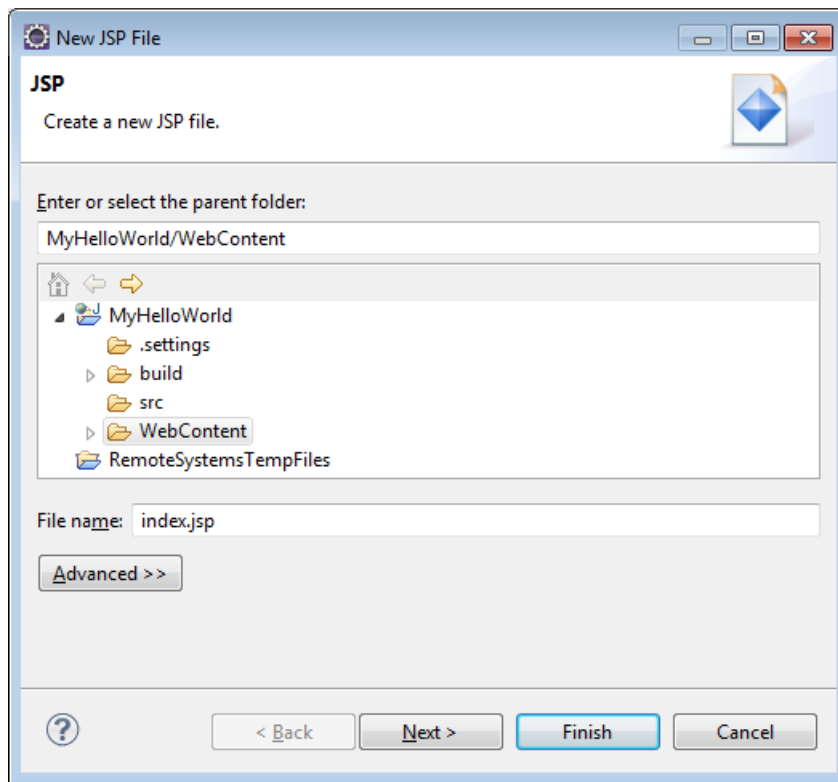
Configuration
Default Configuration Modify...
The default configuration provides a good starting point. Additional facets can later be installed to add new functionality to the project.

EAR membership
☒ Add project to an EAR
EAR project name: EAR New Project...

Working sets
☒ Add project to working sets
Working sets: Select...

< Back Next > Finish Cancel

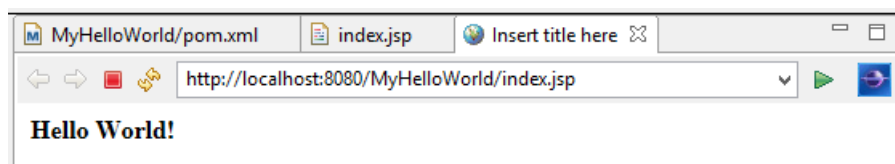
2. Enter a project name in the **Project name** box, and in the **Target Runtime** drop-down menu, optionally select a value (e.g. Apache Tomcat v7.0), and then click **Finish**. Selecting a target runtime enables you to run your project locally through Eclipse.
3. In Eclipse, in the Project Explorer view, expand your project. Right-click **WebContent**, click **New**, and then click **JSP File**.
4. In the **New JSP File** dialog box, name the file `index.jsp`. Keep the parent folder as **WebContent**, as shown in the following illustration, and then click **Next**.



5. In the **Select JSP Template** dialog box, for the purpose of this tutorial select **New JSP File (html)**, and then click **Finish**.
6. When the `index.jsp` file opens in Eclipse, add text to display **Hello World!** within the existing element. Your updated content should look like the following code:

```
<body>
    <% out.println("Hello World!"); %>
</body>
```

7. Save the `index.jsp` file.
8. If you set a target runtime in step 2, you can click **Project** and then **Run** to run your JSP application locally:

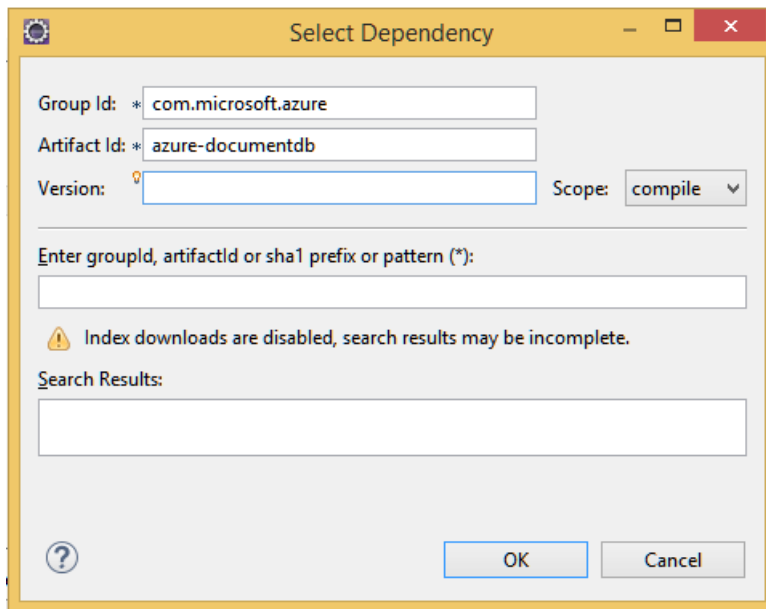


Step 3: Install the DocumentDB Java SDK

The easiest way to pull in the DocumentDB Java SDK and its dependencies is through [Apache Maven](#).

To do this, you will need to convert your project to a maven project by completing the following steps:

1. Right-click your project in the Project Explorer, click **Configure**, click **Convert to Maven Project**.
2. In the **Create new POM** window, accept the defaults and click **Finish**.
3. In **Project Explorer**, open the `pom.xml` file.
4. On the **Dependencies** tab, in the **Dependencies** pane, click **Add**.
5. In the **Select Dependency** window, do the following:
 - In the **GroupId** box, enter `com.microsoft.azure`.
 - In the **Artifact Id** box enter `azure-documentdb`.
 - In the **Version** box enter `1.5.1`.



Or add the dependency XML for GroupId and ArtifactId directly to the pom.xml via a text editor:

```
com.microsoft.azure azure-documentdb 1.9.1
```

6. Click **Ok** and Maven will install the DocumentDB Java SDK.
7. Save the pom.xml file.

Step 4: Using the DocumentDB service in a Java application

1. First, let's define the TodoItem object:

```
@Data
@Builder
public class TodoItem {
    private String category;
    private boolean complete;
    private String id;
    private String name;
}
```

In this project, we are using [Project Lombok](#) to generate the constructor, getters, setters, and a builder. Alternatively, you can write this code manually or have the IDE generate it.

2. To invoke the DocumentDB service, you must instantiate a new **DocumentClient**. In general, it is best to reuse the **DocumentClient** - rather than construct a new client for each subsequent request. We can reuse the client by wrapping the client in a **DocumentClientFactory**. This is also where you need to paste the URI and PRIMARY KEY value you saved to your clipboard in [step 1](#). Replace [YOUR_ENDPOINT_HERE] with your URI and replace [YOUR_KEY_HERE] with your PRIMARY KEY.

```
private static final String HOST = "[YOUR_ENDPOINT_HERE]";
private static final String MASTER_KEY = "[YOUR_KEY_HERE]";

private static DocumentClient documentClient = new DocumentClient(HOST, MASTER_KEY,
    ConnectionPolicy.GetDefault(), ConsistencyLevel.Session);

public static DocumentClient getDocumentClient() {
    return documentClient;
}
```

3. Now let's create a Data Access Object (DAO) to abstract persisting our Todo items to DocumentDB.

In order to save Todo items to a collection, the client needs to know which database and collection to

persist to (as referenced by self-links). In general, it is best to cache the database and collection when possible to avoid additional round-trips to the database.

The following code illustrates how to retrieve our database and collection, if it exists, or create a new one if it doesn't exist:

```
public class DocDbDao implements TodoDao {
    // The name of our database.
    private static final String DATABASE_ID = "TodoDB";

    // The name of our collection.
    private static final String COLLECTION_ID = "TodoCollection";

    // The DocumentDB Client
    private static DocumentClient documentClient = DocumentClientFactory
        .getDocumentClient();

    // Cache for the database object, so we don't have to query for it to
    // retrieve self links.
    private static Database databaseCache;

    // Cache for the collection object, so we don't have to query for it to
    // retrieve self links.
    private static DocumentCollection collectionCache;

    private Database getTodoDatabase() {
        if (databaseCache == null) {
            // Get the database if it exists
            List<Database> databaseList = documentClient
                .queryDatabases(
                    "SELECT * FROM root r WHERE r.id='" + DATABASE_ID
                        + "'", null).getQueryIterable().toList();

            if (databaseList.size() > 0) {
                // Cache the database object so we won't have to query for it
                // later to retrieve the selfLink.
                databaseCache = databaseList.get(0);
            } else {
                // Create the database if it doesn't exist.
                try {
                    Database databaseDefinition = new Database();
                    databaseDefinition.setId(DATABASE_ID);

                    databaseCache = documentClient.createDatabase(
                        databaseDefinition, null).getResource();
                } catch (DocumentClientException e) {
                    // TODO: Something has gone terribly wrong - the app wasn't
                    // able to query or create the collection.
                    // Verify your connection, endpoint, and key.
                    e.printStackTrace();
                }
            }
        }

        return databaseCache;
    }

    private DocumentCollection getTodoCollection() {
        if (collectionCache == null) {
            // Get the collection if it exists.
            List<DocumentCollection> collectionList = documentClient
                .queryCollections(
                    getTodoDatabase().getSelfLink(),
                    "SELECT * FROM root r WHERE r.id='" + COLLECTION_ID
                        + "'", null).getQueryIterable().toList();

            if (collectionList.size() > 0) {
                // Cache the collection object so we won't have to query for it
            }
        }
    }
}
```

```

        // Cache the collection object so we won't have to query for it
        // later to retrieve the selfLink.
        collectionCache = collectionList.get(0);
    } else {
        // Create the collection if it doesn't exist.
        try {
            DocumentCollection collectionDefinition = new DocumentCollection();
            collectionDefinition.setId(COLLECTION_ID);

            collectionCache = documentClient.createCollection(
                getTodoDatabase().getSelfLink(),
                collectionDefinition, null).getResource();
        } catch (DocumentClientException e) {
            // TODO: Something has gone terribly wrong - the app wasn't
            // able to query or create the collection.
            // Verify your connection, endpoint, and key.
            e.printStackTrace();
        }
    }
}

return collectionCache;
}
}

```

4. The next step is to write some code to persist the `TodoItems` in to the collection. In this example, we will use [Gson](#) to serialize and de-serialize `TodoItem` Plain Old Java Objects (POJOs) to JSON documents. [Jackson](#) or your own custom serializer are also great alternatives for serializing POJOs.

```

// We'll use Gson for POJO <=> JSON serialization for this example.
private static Gson gson = new Gson();

@Override
public TodoItem createTodoItem(TodoItem todoItem) {
    // Serialize the TodoItem as a JSON Document.
    Document todoItemDocument = new Document(gson.toJson(todoItem));

    // Annotate the document as a TodoItem for retrieval (so that we can
    // store multiple entity types in the collection).
    todoItemDocument.set("entityType", "todoItem");

    try {
        // Persist the document using the DocumentClient.
        todoItemDocument = documentClient.createDocument(
            getTodoCollection().getSelfLink(), todoItemDocument, null,
            false).getResource();
    } catch (DocumentClientException e) {
        e.printStackTrace();
        return null;
    }

    return gson.fromJson(todoItemDocument.toString(), TodoItem.class);
}

```

5. Like `DocumentDB` databases and collections, documents are also referenced by self-links. The following helper function lets us retrieve documents by another attribute (e.g. "id") rather than self-link:

```

private Document getDocumentById(String id) {
    // Retrieve the document using the DocumentClient.
    List<Document> documentList = documentClient
        .queryDocuments(getTodoCollection().getSelfLink(),
            "SELECT * FROM root r WHERE r.id='" + id + "'", null)
        .getQueryIterable().toList();

    if (documentList.size() > 0) {
        return documentList.get(0);
    } else {
        return null;
    }
}

```

6. We can use the helper method in step 5 to retrieve a TodoItem JSON document by id and then deserialize it to a POJO:

```

@Override
public TodoItem readTodoItem(String id) {
    // Retrieve the document by id using our helper method.
    Document todoItemDocument = getDocumentById(id);

    if (todoItemDocument != null) {
        // De-serialize the document in to a TodoItem.
        return gson.fromJson(todoItemDocument.toString(), TodoItem.class);
    } else {
        return null;
    }
}

```

7. We can also use the DocumentClient to get a collection or list of TodoItems using DocumentDB SQL:

```

@Override
public List<TodoItem> readTodoItems() {
    List<TodoItem> todoItems = new ArrayList<TodoItem>();

    // Retrieve the TodoItem documents
    List<Document> documentList = documentClient
        .queryDocuments(getTodoCollection().getSelfLink(),
            "SELECT * FROM root r WHERE r.entityType = 'todoItem'",
            null).getQueryIterable().toList();

    // De-serialize the documents in to TodoItems.
    for (Document todoItemDocument : documentList) {
        todoItems.add(gson.fromJson(todoItemDocument.toString(),
            TodoItem.class));
    }

    return todoItems;
}

```

8. There are many ways to update a document with the DocumentClient. In our Todo list application, we want to be able to toggle whether a TodoItem is complete. This can be achieved by updating the "complete" attribute within the document:

```

@Override
public TodoItem updateTodoItem(String id, boolean isComplete) {
    // Retrieve the document from the database
    Document todoItemDocument = getDocumentById(id);

    // You can update the document as a JSON document directly.
    // For more complex operations - you could de-serialize the document in
    // to a POJO, update the POJO, and then re-serialize the POJO back in to
    // a document.
    todoItemDocument.set("complete", isComplete);

    try {
        // Persist/replace the updated document.
        todoItemDocument = documentClient.replaceDocument(todoItemDocument,
            null).getResource();
    } catch (DocumentClientException e) {
        e.printStackTrace();
        return null;
    }

    return gson.fromJson(todoItemDocument.toString(), TodoItem.class);
}

```

9. Finally, we want the ability to delete a TodoItem from our list. To do this, we can use the helper method we wrote earlier to retrieve the self-link and then tell the client to delete it:

```

@Override
public boolean deleteTodoItem(String id) {
    // DocumentDB refers to documents by self link rather than id.

    // Query for the document to retrieve the self link.
    Document todoItemDocument = getDocumentById(id);

    try {
        // Delete the document by self link.
        documentClient.deleteDocument(todoItemDocument.getSelfLink(), null);
    } catch (DocumentClientException e) {
        e.printStackTrace();
        return false;
    }

    return true;
}

```

Step 5: Wiring the rest of the of Java application development project together

Now that we've finished the fun bits - all that left is to build a quick user interface and wire it up to our DAO.

1. First, let's start with building a controller to call our DAO:

```

public class TodoItemController {
    public static TodoItemController getInstance() {
        if (todoItemController == null) {
            todoItemController = new TodoItemController(TodoDaoFactory.getDao());
        }
        return todoItemController;
    }

    private static TodoItemController todoItemController;

    private final TodoDao todoDao;

    TodoItemController(TodoDao todoDao) {
        this.todoDao = todoDao;
    }

    public TodoItem createTodoItem(@NonNull String name,
                                   @NonNull String category, boolean isComplete) {
        TodoItem todoItem = TodoItem.builder().name(name).category(category)
            .complete(isComplete).build();
        return todoDao.createTodoItem(todoItem);
    }

    public boolean deleteTodoItem(@NonNull String id) {
        return todoDao.deleteTodoItem(id);
    }

    public TodoItem getTodoItemById(@NonNull String id) {
        return todoDao.readTodoItem(id);
    }

    public List<TodoItem> getTodoItems() {
        return todoDao.readTodoItems();
    }

    public TodoItem updateTodoItem(@NonNull String id, boolean isComplete) {
        return todoDao.updateTodoItem(id, isComplete);
    }
}

```

In a more complex application, the controller may house complicated business logic on top of the DAO.

2. Next, we'll create a servlet to route HTTP requests to the controller:

```

public class TodoServlet extends HttpServlet {
    // API Keys
    public static final String API_METHOD = "method";

    // API Methods
    public static final String CREATE_TODO_ITEM = "createTodoItem";
    public static final String GET_TODO_ITEMS = "getTodoItems";
    public static final String UPDATE_TODO_ITEM = "updateTodoItem";

    // API Parameters
    public static final String TODO_ITEM_ID = "todoItemId";
    public static final String TODO_ITEM_NAME = "todoItemName";
    public static final String TODO_ITEM_CATEGORY = "todoItemCategory";
    public static final String TODO_ITEM_COMPLETE = "todoItemComplete";

    public static final String MESSAGE_ERROR_INVALID_METHOD = "{ 'error': 'Invalid method' }";

    private static final long serialVersionUID = 1L;
    private static final Gson gson = new Gson();

    @Override
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {

        String apiResponse = MESSAGE_ERROR_INVALID_METHOD;

        TodoItemController todoItemController = TodoItemController
            .getInstance();

        String id = request.getParameter(TODO_ITEM_ID);
        String name = request.getParameter(TODO_ITEM_NAME);
        String category = request.getParameter(TODO_ITEM_CATEGORY);
        boolean isComplete = StringUtils.equalsIgnoreCase("true",
            request.getParameter(TODO_ITEM_COMPLETE)) ? true : false;

        switch (request.getParameter(API_METHOD)) {
            case CREATE_TODO_ITEM:
                apiResponse = gson.toJson(todoItemController.createTodoItem(name,
                    category, isComplete));
                break;
            case GET_TODO_ITEMS:
                apiResponse = gson.toJson(todoItemController.getTodoItems());
                break;
            case UPDATE_TODO_ITEM:
                apiResponse = gson.toJson(todoItemController.updateTodoItem(id,
                    isComplete));
                break;
            default:
                break;
        }

        response.getWriter().println(apiResponse);
    }

    @Override
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        doGet(request, response);
    }
}

```

3. We'll need a Web User Interface to display to the user. Let's re-write the index.jsp we created earlier:

```

<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <meta http-equiv="X-UA-Compatible" content="IE=edge;" />

```

```

<title>Azure DocumentDB Java Sample</title>

<!-- Bootstrap -->
<link href="//ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/css/bootstrap.min.css" rel="stylesheet">

<style>
  /* Add padding to body for fixed nav bar */
  body {
    padding-top: 50px;
  }
</style>
</head>
<body>
  <!-- Nav Bar -->
  <div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
    <div class="container">
      <div class="navbar-header">
        <a class="navbar-brand" href="#">My Tasks</a>
      </div>
    </div>
  </div>

  <!-- Body -->
  <div class="container">
    <h1>My ToDo List</h1>

    <hr/>

    <!-- The ToDo List -->
    <div class = "todoList">
      <table class="table table-bordered table-striped" id="todoItems">
        <thead>
          <tr>
            <th>Name</th>
            <th>Category</th>
            <th>Complete</th>
          </tr>
        </thead>
        <tbody>
        </tbody>
      </table>

      <!-- Update Button -->
      <div class="todoUpdatePanel">
        <form class="form-horizontal" role="form">
          <button type="button" class="btn btn-primary">Update Tasks</button>
        </form>
      </div>

    </div>

    <hr/>

    <!-- Item Input Form -->
    <div class="todoForm">
      <form class="form-horizontal" role="form">
        <div class="form-group">
          <label for="inputItemName" class="col-sm-2">Task Name</label>
          <div class="col-sm-10">
            <input type="text" class="form-control" id="inputItemName" placeholder="Enter name">
          </div>
        </div>

        <div class="form-group">
          <label for="inputItemCategory" class="col-sm-2">Task Category</label>
          <div class="col-sm-10">
            <input type="text" class="form-control" id="inputItemCategory" placeholder="Enter category">
          </div>
        </div>
      </form>
    </div>
  </div>

```

```

        <button type="button" class="btn btn-primary">Add Task</button>
    </form>
</div>

</div>

<!-- Placed at the end of the document so the pages load faster -->
<script src="//ajax.aspnetcdn.com/ajax/jquery/jquery-2.1.1.min.js"></script>
<script src="//ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/bootstrap.min.js"></script>
<script src="assets/todo.js"></script>
</body>
</html>

```

4. And finally, write some client-side Javascript to tie the web user interface and the servlet together:

```

var todoApp = {
    /*
     * API methods to call Java backend.
     */
    apiEndpoint: "api",

    createTodoItem: function(name, category, isComplete) {
        $.post(todoApp.apiEndpoint, {
            "method": "createTodoItem",
            "todoItemName": name,
            "todoItemCategory": category,
            "todoItemComplete": isComplete
        },
        function(data) {
            var todoItem = data;
            todoApp.addTodoItemToTable(todoItem.id, todoItem.name, todoItem.category, todoItem.complete);
        },
        "json");
    },

    getTodoItems: function() {
        $.post(todoApp.apiEndpoint, {
            "method": "getTodoItems"
        },
        function(data) {
            var todoItemArr = data;
            $.each(todoItemArr, function(index, value) {
                todoApp.addTodoItemToTable(value.id, value.name, value.category, value.complete);
            });
        },
        "json");
    },

    updateTodoItem: function(id, isComplete) {
        $.post(todoApp.apiEndpoint, {
            "method": "updateTodoItem",
            "todoItemId": id,
            "todoItemComplete": isComplete
        },
        function(data) {},
        "json");
    },

    /*
     * UI Methods
     */
    addTodoItemToTable: function(id, name, category, isComplete) {
        var rowColor = isComplete ? "active" : "warning";

        todoApp.ui_table().append($("<tr>")
            .append($("<td>").text(name))

```

```

        .append($("<td>").text(category))
        .append($("<td>")
            .append($("<input>")
                .attr("type", "checkbox")
                .attr("id", id)
                .attr("checked", isComplete)
                .attr("class", "isComplete")
            ))
        .addClass(rowColor)
    );
},

/*
 * UI Bindings
 */
bindCreateButton: function() {
    todoApp.ui_createButton().click(function() {
        todoApp.createTodoItem(todoApp.ui_createNameInput().val(), todoApp.ui_createCategoryInput().val(),
false);
        todoApp.ui_createNameInput().val("");
        todoApp.ui_createCategoryInput().val("");
    });
},

bindUpdateButton: function() {
    todoApp.ui_updateButton().click(function() {
        // Disable button temporarily.
        var myButton = $(this);
        var originalText = myButton.text();
        $(this).text("Updating...");
        $(this).prop("disabled", true);

        // Call api to update todo items.
        $.each(todoApp.ui_updateId(), function(index, value) {
            todoApp.updateTodoItem(value.name, value.value);
            $(value).remove();
        });

        // Re-enable button.
        setTimeout(function() {
            myButton.prop("disabled", false);
            myButton.text(originalText);
        }, 500);
    });
},

bindUpdateCheckboxes: function() {
    todoApp.ui_table().on("click", ".isComplete", function(event) {
        var checkboxElement = $(event.currentTarget);
        var rowElement = $(event.currentTarget).parents('tr');
        var id = checkboxElement.attr('id');
        var isComplete = checkboxElement.is(':checked');

        // Toggle table row color
        if (isComplete) {
            rowElement.addClass("active");
            rowElement.removeClass("warning");
        } else {
            rowElement.removeClass("active");
            rowElement.addClass("warning");
        }

        // Update hidden inputs for update panel.
        todoApp.ui_updateForm().children("input[name='" + id + "']").remove();

        todoApp.ui_updateForm().append($("<input>")
            .attr("type", "hidden")
            .attr("class", "updateComplete")
            .attr("name", id)

```

```

        .attr("value", isComplete));

    });
},

/*
 * UI Elements
 */
ui_createNameInput: function() {
    return $(".todoForm #inputItemName");
},

ui_createCategoryInput: function() {
    return $(".todoForm #inputItemCategory");
},

ui_createButton: function() {
    return $(".todoForm button");
},

ui_table: function() {
    return $(".todoList table tbody");
},

ui_updateButton: function() {
    return $(".todoUpdatePanel button");
},

ui_updateForm: function() {
    return $(".todoUpdatePanel form");
},

ui_updateId: function() {
    return $(".todoUpdatePanel .updateComplete");
},

/*
 * Install the TodoApp
 */
install: function() {
    todoApp.bindCreateButton();
    todoApp.bindUpdateButton();
    todoApp.bindUpdateCheckboxes();

    todoApp.getTodoItems();
}
};

$(document).ready(function() {
    todoApp.install();
});

```

5. Awesome! Now all that's left is to test the application. Run the application locally, and add some Todo items by filling in the item name and category and clicking **Add Task**.
6. Once the item appears, you can update whether it's complete by toggling the checkbox and clicking **Update Tasks**.

Step 6: Deploy your Java application to Azure Websites

Azure Websites makes deploying Java Applications as simple as exporting your application as a WAR file and either uploading it via source control (e.g. GIT) or FTP.

1. To export your application as a WAR, right-click on your project in **Project Explorer**, click **Export**, and then click **WAR File**.
2. In the **WAR Export** window, do the following:

- In the Web project box, enter azure-documentdb-java-sample.
 - In the Destination box, choose a destination to save the WAR file.
 - Click **Finish**.
3. Now that you have a WAR file in hand, you can simply upload it to your Azure Website's **webapps** directory. For instructions on uploading the file, see [Adding an application to your Java website on Azure](#).

Once the WAR file is uploaded to the webapps directory, the runtime environment will detect that you've added it and will automatically load it.

4. To view your finished product, navigate to http://YOUR_SITE_NAME.azurewebsites.net/azure-documentdb-java-sample/ and start adding your tasks!

Get the project from GitHub

All the samples in this tutorial are included in the [todo](#) project on GitHub. To import the todo project into Eclipse, ensure you have the software and resources listed in the [Prerequisites](#) section, then do the following:

1. Install [Project Lombok](#). Lombok is used to generate constructors, getters, setters in the project. Once you have downloaded the lombok.jar file, double-click it to install it or install it from the command line.
2. If Eclipse is open, close it and restart it to load Lombok.
3. In Eclipse, on the **File** menu, click **Import**.
4. In the **Import** window, click **Git**, click **Projects from Git**, and then click **Next**.
5. On the **Select Repository Source** screen, click **Clone URI**.
6. On the **Source Git Repository** screen, in the URI box, enter <https://github.com/Azure-Samples/documentdb-java-todo-app.git>, and then click **Next**.
7. On the **Branch Selection** screen, ensure that **master** is selected, and then click **Next**.
8. On the **Local Destination** screen, click **Browse** to select a folder where the repository can be copied, and then click **Next**.
9. On the **Select a wizard to use for importing projects** screen, ensure that **Import existing projects** is selected, and then click **Next**.
10. On the **Import Projects** screen, unselect the **DocumentDB** project, and then click **Finish**. The DocumentDB project contains the DocumentDB Java SDK, which we will add as a dependency instead.
11. In **Project Explorer**, navigate to azure-documentdb-java-sample\src\com.microsoft.azure.documentdb.sample.dao\DocumentClientFactory.java and replace the HOST and MASTER_KEY values with the URI and PRIMARY KEY for your DocumentDB account, and then save the file. For more information, see [Step 1. Create a DocumentDB database account](#).
12. In **Project Explorer**, right-click the **azure-documentdb-java-sample**, click **Build Path**, and then click **Configure Build Path**.
13. On the **Java Build Path** screen, in the right pane, select the **Libraries** tab, and then click **Add External JARs**. Navigate to the location of the lombok.jar file, and click **Open**, and then click **OK**.
14. Use step 12 to open the **Properties** window again, and then in the left pane click **Targeted Runtimes**.
15. On the **Targeted Runtimes** screen, click **New**, select **Apache Tomcat v7.0**, and then click **OK**.
16. Use step 12 to open the **Properties** window again, and then in the left pane click **Project Facets**.
17. On the **Project Facets** screen, select **Dynamic Web Module** and **Java**, and then click **OK**.
18. On the **Servers** tab at the bottom of the screen, right-click **Tomcat v7.0 Server at localhost** and then click **Add and Remove**.
19. On the **Add and Remove** window, move **azure-documentdb-java-sample** to the **Configured** box, and then click **Finish**.
20. In the **Server** tab, right-click **Tomcat v7.0 Server at localhost**, and then click **Restart**.
21. In a browser, navigate to <http://localhost:8080/azure-documentdb-java-sample/> and start adding to your task

list. Note that if you changed your default port values, change 8080 to the value you selected.

22. To deploy your project to an Azure web site, see [Step 6. Deploy your application to Azure Websites](#).

Python Flask Web Application Development with DocumentDB

11/22/2016 • 12 min to read • [Edit on GitHub](#)

Contributors

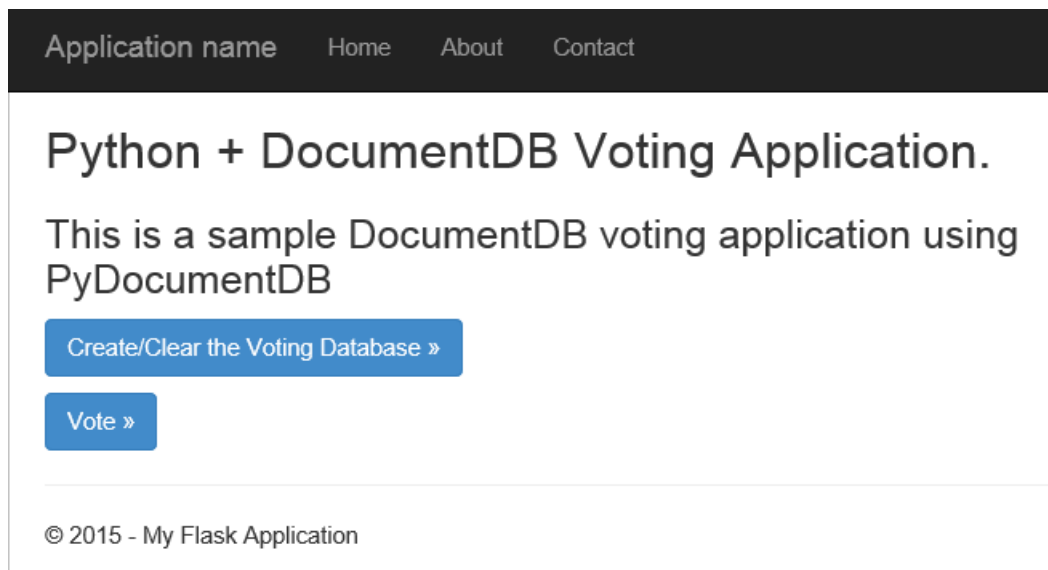
Syam Nair • mimig • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Andrew Hoh • Andrew Liu • Ryan CrawCour
• Tom Dykstra • Sigrid Elenga

This tutorial shows you how to use Azure DocumentDB to store and access data from a Python web application hosted on Azure and presumes that you have some prior experience using Python and Azure websites.

This database tutorial covers:

1. Creating and provisioning a DocumentDB account.
2. Creating a Python MVC application.
3. Connecting to and using Azure DocumentDB from your web application.
4. Deploying the web application to Azure Websites.

By following this tutorial, you will build a simple voting application that allows you to vote for a poll.



Database tutorial prerequisites

Before following the instructions in this article, you should ensure that you have the following installed:

- An active Azure account. If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see [Azure Free Trial](#).

OR

A local installation of the [Azure DocumentDB Emulator](#).

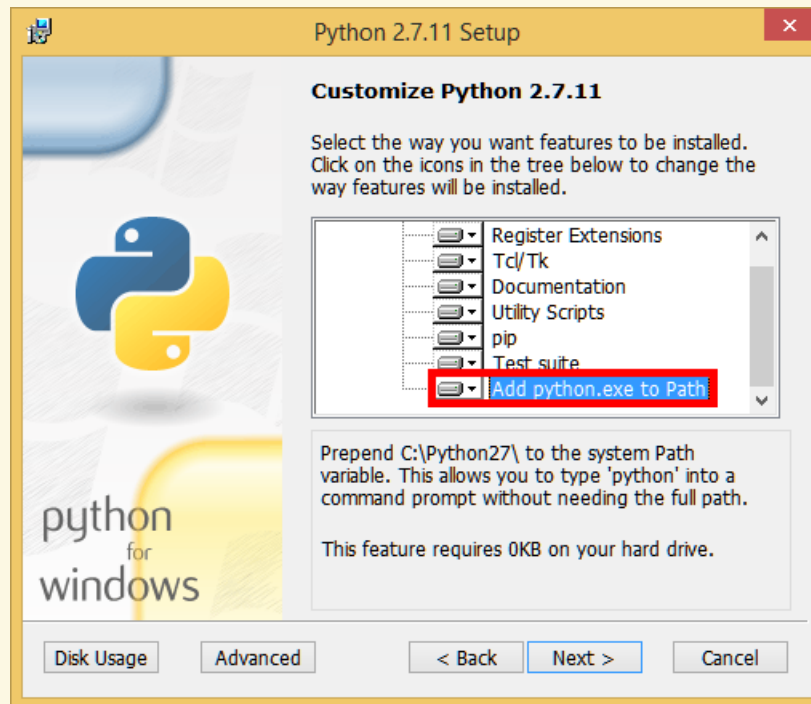
- [Visual Studio 2013](#) or higher, or [Visual Studio Express](#)(), which is the free version. The instructions in this tutorial are written specifically for Visual Studio 2015.
- Python Tools for Visual Studio from [GitHub](#). This tutorial uses Python Tools for VS 2015.
- Azure Python SDK for Visual Studio, version 2.4 or higher available from [azure.com](#). We used Microsoft Azure

SDK for Python 2.7.

- Python 2.7 from python.org. We used Python 2.7.11.

IMPORTANT

If you are installing Python 2.7 for the first time, ensure that in the Customize Python 2.7.11 screen, you select **Add python.exe to Path**.

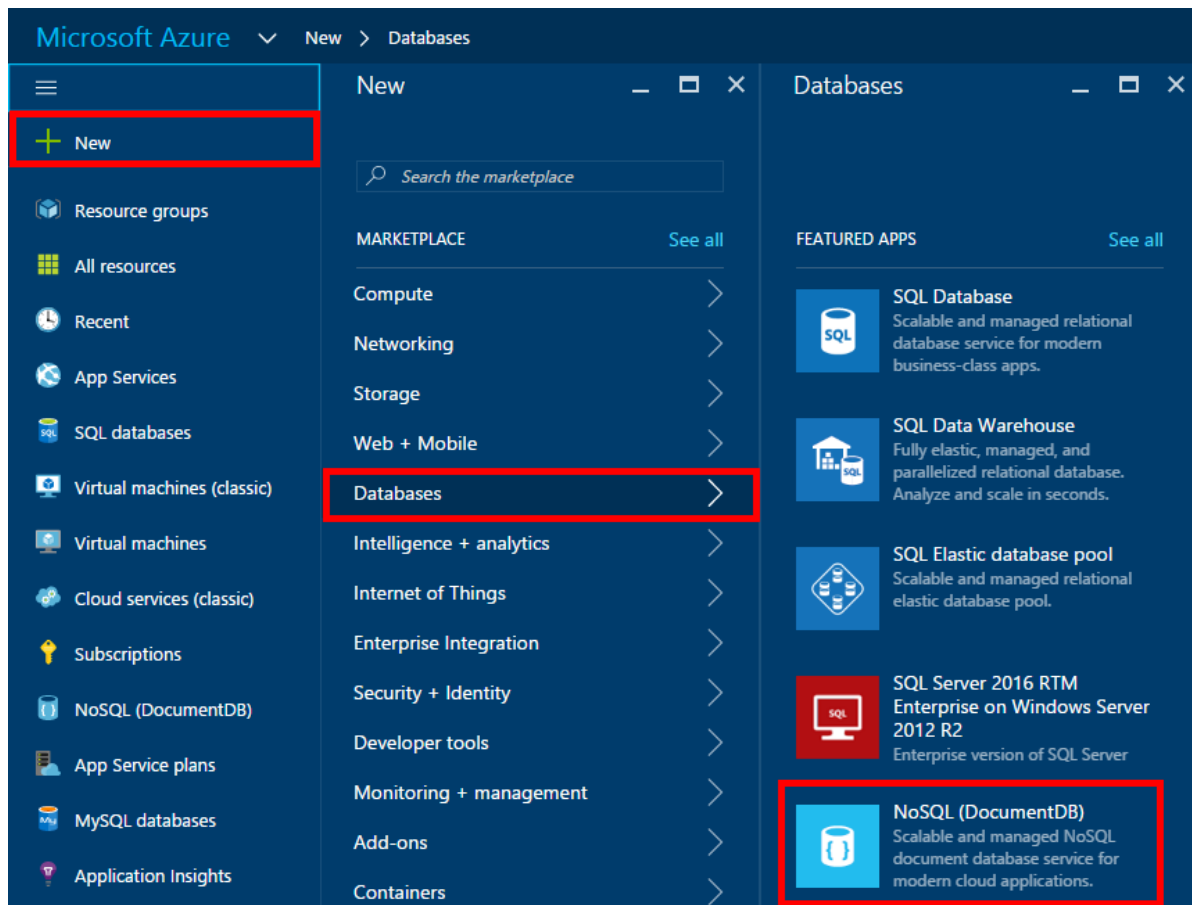


- Microsoft Visual C++ Compiler for Python 2.7 from the [Microsoft Download Center](https://www.microsoft.com/download/center/).

Step 1: Create a DocumentDB database account

Let's start by creating a DocumentDB account. If you already have an account or if you are using the DocumentDB Emulator for this tutorial, you can skip to [Step 2: Create a new Python Flask web application](#).

1. Sign in to the [Azure portal](#).
2. In the Jumpbar, click **New**, click **Databases**, and then click **NoSQL (DocumentDB)**.



3. In the **New** account blade, specify the desired configuration for the DocumentDB account.

NoSQL (DocumentDB)
New account

* ID
contosoacct ✓
documents.azure.com

NoSQL API ⓘ
DocumentDB MongoDB

* Subscription
Visual Studio Ultimate with MSDN ▼

* Resource Group ⓘ
☒ Create new ☐ Use existing
contosoacct ✓

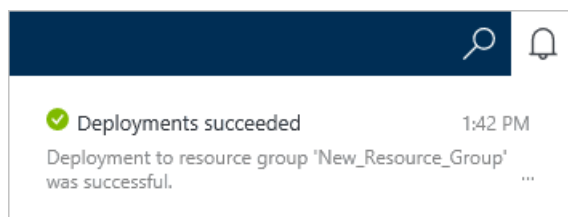
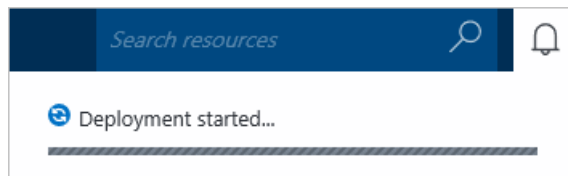
* Location
West US ▼

☐ Pin to dashboard

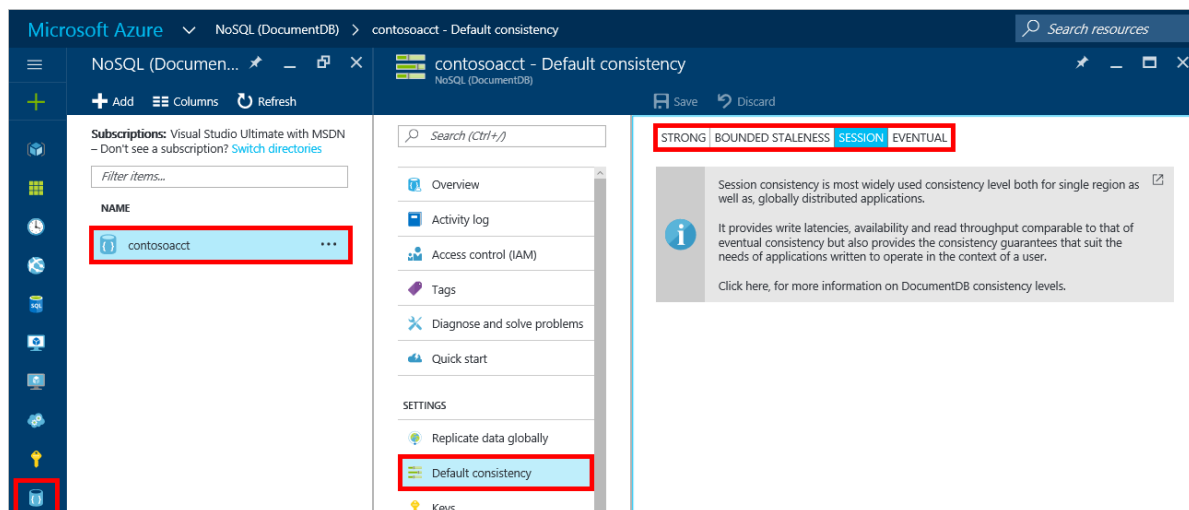
Create [Automation options](#)

- In the **ID** box, enter a name to identify the DocumentDB account. When the **ID** is validated, a green check mark appears in the **ID** box. The **ID** value becomes the host name within the URI. The **ID** may contain only lowercase letters, numbers, and the '-' character, and must be between 3 and 50 characters. Note that *documents.azure.com* is appended to the endpoint name you choose, the result of which becomes your DocumentDB account endpoint.

- In the **NoSQL API** box, select **DocumentDB**.
 - For **Subscription**, select the Azure subscription that you want to use for the DocumentDB account. If your account has only one subscription, that account is selected by default.
 - In **Resource Group**, select or create a resource group for your DocumentDB account. By default, a new resource group is created. For more information, see [Using the Azure portal to manage your Azure resources](#).
 - Use **Location** to specify the geographic location in which to host your DocumentDB account.
4. Once the new DocumentDB account options are configured, click **Create**. To check the status of the deployment, check the Notifications hub.



5. After the DocumentDB account is created, it is ready for use with the default settings. To review the default settings, click the **NoSQL (DocumentDB)** icon on the Jumpbar, click your new account, and then click **Default Consistency** in the Resource Menu.



The default consistency of the DocumentDB account is set to **Session**. You can adjust the default consistency by clicking **Default Consistency** in the resource menu. To learn more about the consistency levels offered by DocumentDB, see [Consistency levels in DocumentDB](#).

We will now walk through how to create a new Python Flask web application from the ground up.

Step 2: Create a new Python Flask web application

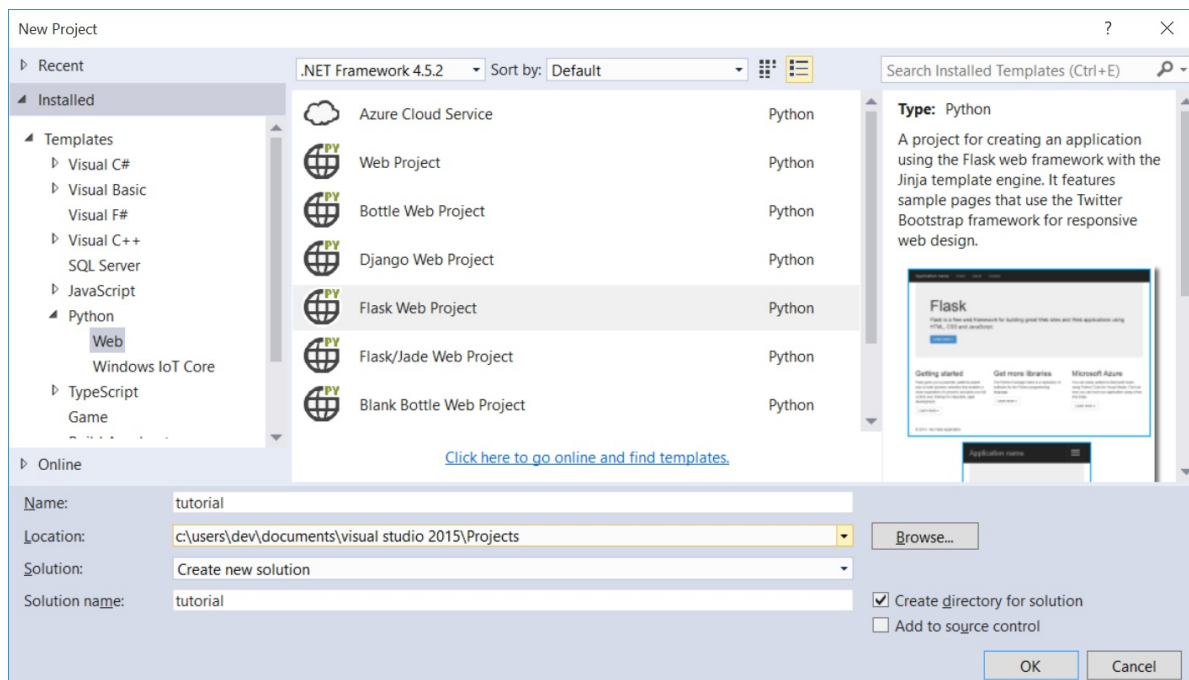
1. In Visual Studio, on the **File** menu, point to **New**, and then click **Project**.

The **New Project** dialog box appears.

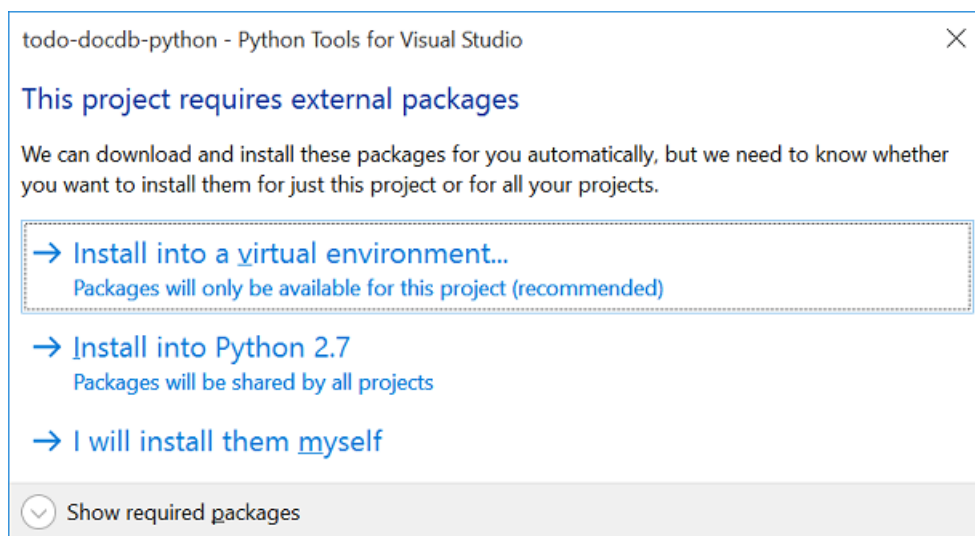
2. In the left pane, expand **Templates** and then **Python**, and then click **Web**.
3. Select **Flask Web Project** in the center pane, then in the **Name** box type **tutorial**, and then click **OK**.

Remember that Python package names should be all lowercase, as described in the [Style Guide for Python Code](#).

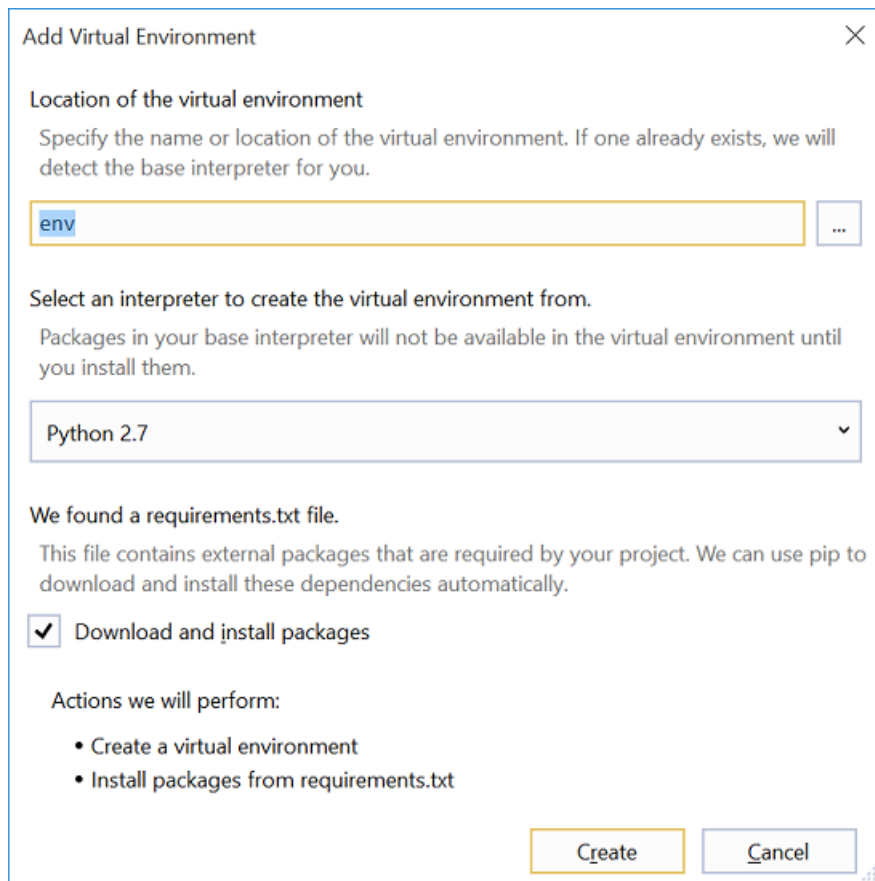
For those new to Python Flask, it is a web application development framework that helps you build web applications in Python faster.



4. In the **Python Tools for Visual Studio** window, click **Install into a virtual environment**.



5. In the **Add Virtual Environment** window, you can accept the defaults and use Python 2.7 as the base environment because PyDocumentDB does not currently support Python 3.x, and then click **Create**. This sets up the required Python virtual environment for your project.



The output window displays

```
Successfully installed Flask-0.10.1 Jinja2-2.8 MarkupSafe-0.23 Werkzeug-0.11.5 itsdangerous-0.24  
'requirements.txt' was installed successfully.
```

when the environment is successfully installed.

Step 3: Modify the Python Flask web application

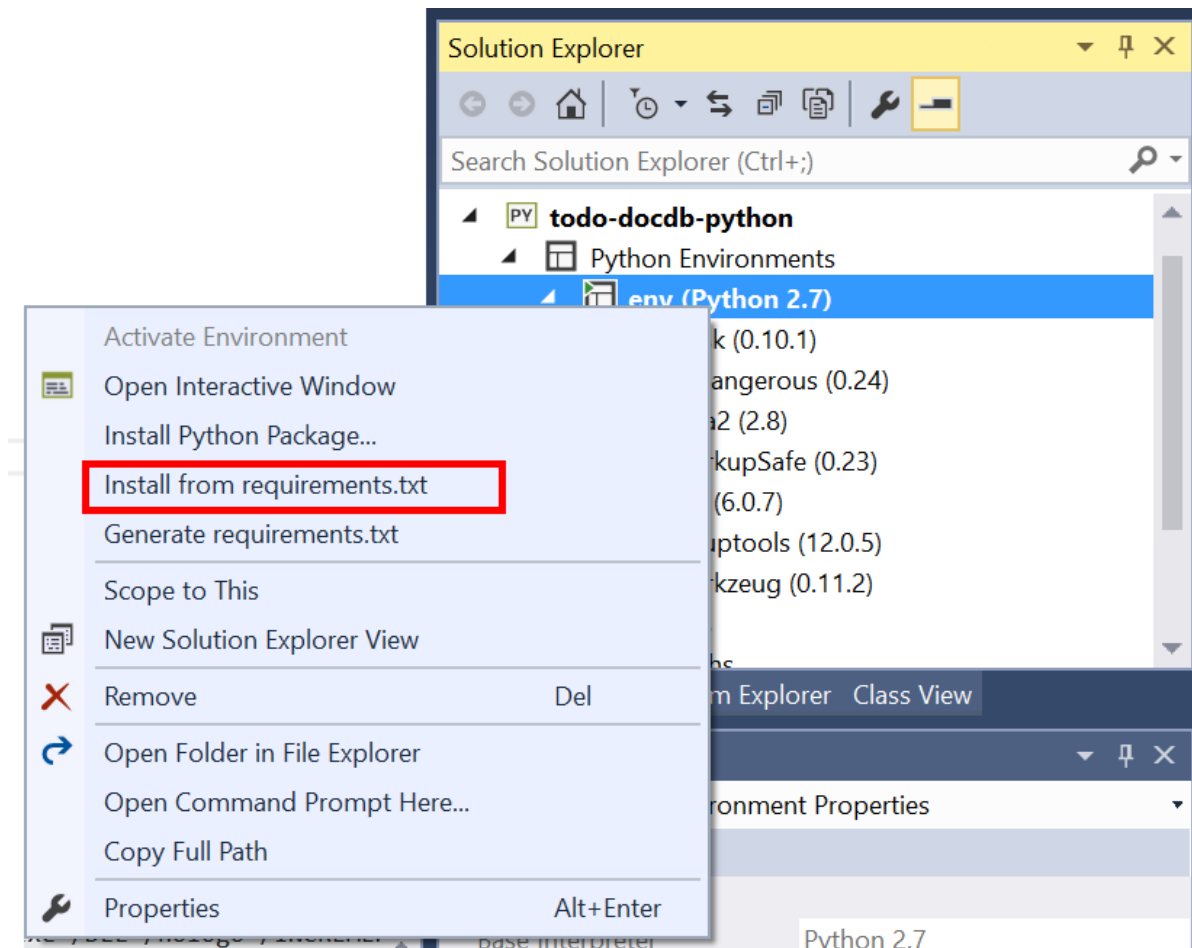
Add the Python Flask packages to your project

After your project is set up, you'll need to add the required Flask packages to your project, including pydocumentdb, the Python package for DocumentDB.

1. In Solution Explorer, open the file named **requirements.txt** and replace the contents with the following:

```
flask==0.9  
flask-mail==0.7.6  
sqlalchemy==0.7.9  
flask-sqlalchemy==0.16  
sqlalchemy-migrate==0.7.2  
flask-whooshalchemy==0.55a  
flask-wtf==0.8.4  
pytz==2013b  
flask-babel==0.8  
flup  
pydocumentdb>=1.0.0
```

2. Save the **requirements.txt** file.
3. In Solution Explorer, right-click **env** and click **Install from requirements.txt**.



After successful installation, the output window displays the following:

```
Successfully installed Babel-2.3.2 Tempita-0.5.2 WTForms-2.1 Whoosh-2.7.4 blinker-1.4 decorator-4.0.9
flask-0.9 flask-babel-0.8 flask-mail-0.7.6 flask-sqlalchemy-0.16 flask-whooshalchemy-0.55a0 flask-wtf-
0.8.4 flup-1.0.2 pydocumentdb-1.6.1 pytz-2013b0 speaklater-1.3 sqlalchemy-0.7.9 sqlalchemy-migrate-0.7.2
```

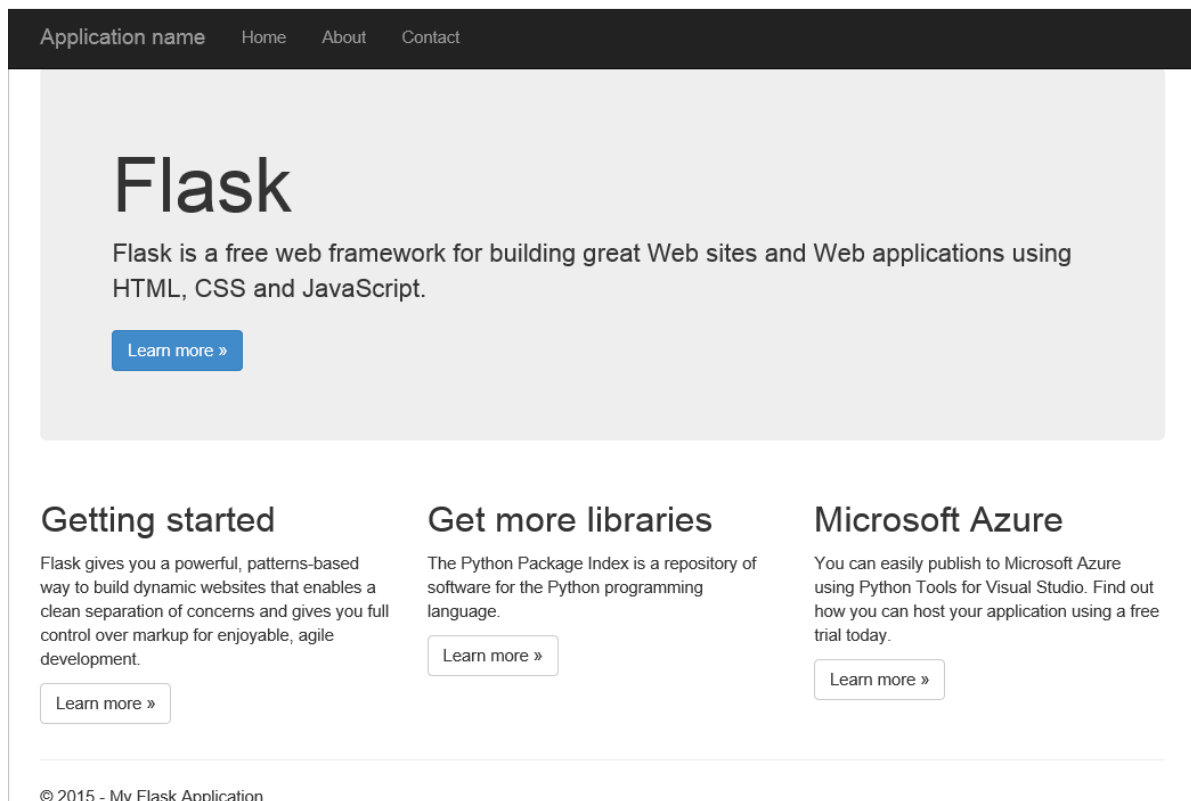
NOTE

In rare cases, you might see a failure in the output window. If this happens, check if the error is related to cleanup. Sometimes the cleanup fails, but the installation will still be successful (scroll up in the output window to verify this). You can check your installation by [Verifying the virtual environment](#). If the installation failed but the verification is successful, it's OK to continue.

Verify the virtual environment

Let's make sure that everything is installed correctly.

1. Build the solution by pressing **Ctrl+Shift+B**.
2. Once the build succeeds, start the website by pressing **F5**. This launches the Flask development server and starts your web browser. You should see the following page.



3. Stop debugging the website by pressing **Shift+F5** in Visual Studio.

Create database, collection, and document definitions

Now let's create your voting application by adding new files and updating others.

1. In Solution Explorer, right-click the **tutorial** project, click **Add**, and then click **New Item**. Select **Empty Python File** and name the file **forms.py**.
2. Add the following code to the **forms.py** file, and then save the file.

```
from flask.ext.wtf import Form
from wtforms import RadioField

class VoteForm(Form):
    deploy_preference = RadioField('Deployment Preference', choices=[
        ('Web Site', 'Web Site'),
        ('Cloud Service', 'Cloud Service'),
        ('Virtual Machine', 'Virtual Machine')], default='Web Site')
```

Add the required imports to views.py

1. In Solution Explorer, expand the **tutorial** folder, and open the **views.py** file.
2. Add the following import statements to the top of the **views.py** file, then save the file. These import DocumentDB's PythonSDK and the Flask packages.

```
from forms import VoteForm
import config
import pydocumentdb.document_client as document_client
```

Create database, collection, and document

- Still in **views.py**, add the following code to the end of the file. This takes care of creating the database used by the form. Do not delete any of the existing code in **views.py**. Simply append this to the end.

```

@app.route('/create')
def create():
    """Renders the contact page."""
    client = document_client.DocumentClient(config.DOCUMENTDB_HOST, {'masterKey': config.DOCUMENTDB_KEY})

    # Attempt to delete the database. This allows this to be used to recreate as well as create
    try:
        db = next((data for data in client.ReadDatabases() if data['id'] == config.DOCUMENTDB_DATABASE))
        client.DeleteDatabase(db['_self'])
    except:
        pass

    # Create database
    db = client.CreateDatabase({ 'id': config.DOCUMENTDB_DATABASE })

    # Create collection
    collection = client.CreateCollection(db['_self'],{ 'id': config.DOCUMENTDB_COLLECTION })

    # Create document
    document = client.CreateDocument(collection['_self'],
        { 'id': config.DOCUMENTDB_DOCUMENT,
          'Web Site': 0,
          'Cloud Service': 0,
          'Virtual Machine': 0,
          'name': config.DOCUMENTDB_DOCUMENT
        })

    return render_template(
        'create.html',
        title='Create Page',
        year=datetime.now().year,
        message='You just created a new database, collection, and document. Your old votes have been deleted')

```

TIP

The **CreateCollection** method takes an optional **RequestOptions** as the third parameter. This can be used to specify the Offer Type for the collection. If no offerType value is supplied, then the collection will be created using the default Offer Type. For more information on DocumentDB Offer Types, see [Performance levels in DocumentDB](#).

Read database, collection, document, and submit form

- Still in **views.py**, add the following code to the end of the file. This takes care of setting up the form, reading the database, collection, and document. Do not delete any of the existing code in **views.py**. Simply append this to the end.

```

@app.route('/vote', methods=['GET', 'POST'])
def vote():
    form = VoteForm()
    replaced_document = {}
    if form.validate_on_submit(): # is user submitted vote
        client = document_client.DocumentClient(config.DOCUMENTDB_HOST, {'masterKey': config.DOCUMENTDB_KEY})

        # Read databases and take first since id should not be duplicated.
        db = next((data for data in client.ReadDatabases() if data['id'] == config.DOCUMENTDB_DATABASE))

        # Read collections and take first since id should not be duplicated.
        coll = next((coll for coll in client.ReadCollections(db['_self']) if coll['id'] ==
config.DOCUMENTDB_COLLECTION))

        # Read documents and take first since id should not be duplicated.
        doc = next((doc for doc in client.ReadDocuments(coll['_self']) if doc['id'] ==
config.DOCUMENTDB_DOCUMENT))

        # Take the data from the deploy_preference and increment our database
        doc[form.deploy_preference.data] = doc[form.deploy_preference.data] + 1
        replaced_document = client.ReplaceDocument(doc['_self'], doc)

        # Create a model to pass to results.html
        class VoteObject:
            choices = dict()
            total_votes = 0

        vote_object = VoteObject()
        vote_object.choices = {
            "Web Site" : doc['Web Site'],
            "Cloud Service" : doc['Cloud Service'],
            "Virtual Machine" : doc['Virtual Machine']
        }
        vote_object.total_votes = sum(vote_object.choices.values())

        return render_template(
            'results.html',
            year=datetime.now().year,
            vote_object = vote_object)

    else :
        return render_template(
            'vote.html',
            title = 'Vote',
            year=datetime.now().year,
            form = form)

```

Create the HTML files

1. In Solution Explorer, in the **tutorial** folder, right click the **templates** folder, click **Add**, and then click **New Item**.
2. Select **HTML Page**, and then in the name box type **create.html**.
3. Repeat steps 1 and 2 to create two additional HTML files: **results.html** and **vote.html**.
4. Add the following code to **create.html** in the `<body>` element. It displays a message stating that we created a new database, collection, and document.

```

{% extends "layout.html" %}
{% block content %}
<h2>{{ title }}</h2>
<h3>{{ message }}</h3>
<p><a href="{{ url_for('vote') }}" class="btn btn-primary btn-large">Vote &raquo;</a></p>
{% endblock %}

```

5. Add the following code to **results.html** in the `<body>` element. It displays the results of the poll.

```
{% extends "layout.html" %}
{% block content %}
<h2>Results of the vote</h2>
    <br />

{% for choice in vote_object.choices %}
<div class="row">
    <div class="col-sm-5">{{choice}}</div>
    <div class="col-sm-5">
        <div class="progress">
            <div class="progress-bar" role="progressbar" aria-valuenow="{{vote_object.choices[choice]}}"
            aria-valuemin="0" aria-valuemax="{{vote_object.total_votes}}"
            style="width: {{(vote_object.choices[choice]/vote_object.total_votes)*100}}%;">
                {{vote_object.choices[choice]}}
            </div>
        </div>
    </div>
</div>
{% endfor %}

<br />
<a class="btn btn-primary" href="{{ url_for('vote') }}">Vote again?</a>
{% endblock %}
```

6. Add the following code to **vote.html** in the `<body>` element. It displays the poll and accepts the votes. On registering the votes, the control is passed over to `views.py` where we will recognize the vote cast and append the document accordingly.

```
{% extends "layout.html" %}
{% block content %}
<h2>What is your favorite way to host an application on Azure?</h2>
<form action="" method="post" name="vote">
    {{form.hidden_tag()}}
    {{form.deploy_preference}}
    <button class="btn btn-primary" type="submit">Vote</button>
</form>
{% endblock %}
```

7. In the **templates** folder, replace the contents of **index.html** with the following. This serves as the landing page for your application.

```
{% extends "layout.html" %}
{% block content %}
<h2>Python + DocumentDB Voting Application.</h2>
<h3>This is a sample DocumentDB voting application using PyDocumentDB</h3>
<p><a href="{{ url_for('create') }}" class="btn btn-primary btn-large">Create/Clear the Voting Database
&raquo;</a></p>
<p><a href="{{ url_for('vote') }}" class="btn btn-primary btn-large">Vote &raquo;</a></p>
{% endblock %}
```

Add a configuration file and change the `__init__.py`

1. In Solution Explorer, right-click the **tutorial** project, click **Add**, click **New Item**, select **Empty Python File**, and then name the file **config.py**. This config file is required by forms in Flask. You can use it to provide a secret key as well. This key is not needed for this tutorial though.
2. Add the following code to `config.py`, you'll need to alter the values of `DOCUMENTDB_HOST` and `DOCUMENTDB_KEY` in the next step.

```
CSRF_ENABLED = True
SECRET_KEY = 'you-will-never-guess'

DOCUMENTDB_HOST = 'https://YOUR_DOCUMENTDB_NAME.documents.azure.com:443/'
DOCUMENTDB_KEY = 'YOUR_SECRET_KEY_ENDING_IN=='

DOCUMENTDB_DATABASE = 'voting database'
DOCUMENTDB_COLLECTION = 'voting collection'
DOCUMENTDB_DOCUMENT = 'voting document'
```

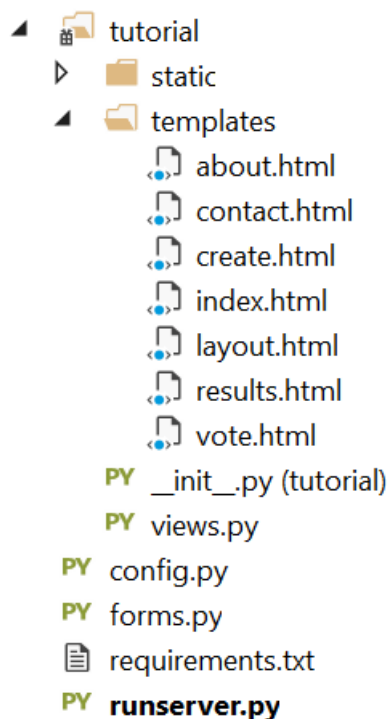
3. In the [Azure portal](#), navigate to the **Keys** blade by clicking **Browse, DocumentDB Accounts**, double-click the name of the account to use, and then click the **Keys** button in the **Essentials** area. In the **Keys** blade, copy the **URI** value and paste it into the **config.py** file, as the value for the **DOCUMENTDB_HOST** property.
4. Back in the Azure portal, in the **Keys** blade, copy the value of the **Primary Key** or the **Secondary Key**, and paste it into the **config.py** file, as the value for the **DOCUMENTDB_KEY** property.
5. In the **__init__.py** file, add the following line.

```
app.config.from_object('config')
```

So that the content of the file is:

```
from flask import Flask
app = Flask(__name__)
app.config.from_object('config')
import tutorial.views
```

6. After adding all the files, Solution Explorer should look like this:



Step 4: Run your web application locally

1. Build the solution by pressing **Ctrl+Shift+B**.
2. Once the build succeeds, start the website by pressing **F5**. You should see the following on your screen.

Application name Home About Contact

Python + DocumentDB Voting Application.

This is a sample DocumentDB voting application using PyDocumentDB

Create/Clear the Voting Database »

Vote »

© 2015 - My Flask Application

3. Click **Create/Clear the Voting Database** to generate the database.

Application name Home About Contact

Create Page.

You just created a new database, collection, and document. Your old votes have been deleted

Vote »

© 2015 - My Flask Application

4. Then, click **Vote** and select your option.

Application name Home About Contact

What is your favorite way to host an application on Azure?

☒ Web Site

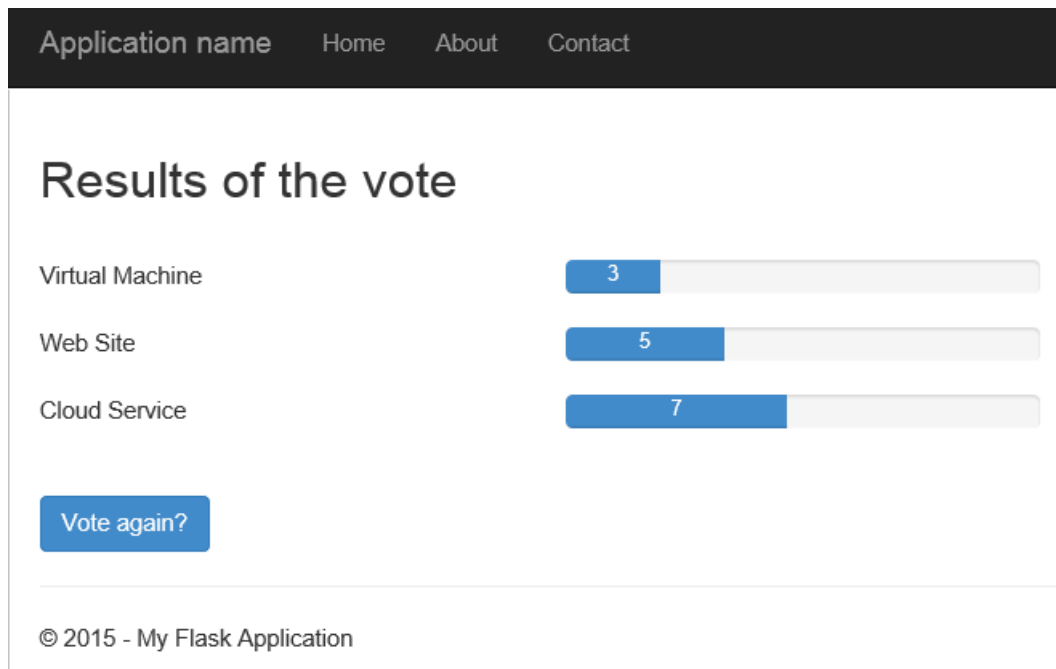
☐ Cloud Service

☐ Virtual Machine

Vote

© 2015 - My Flask Application

5. For every vote you cast, it increments the appropriate counter.

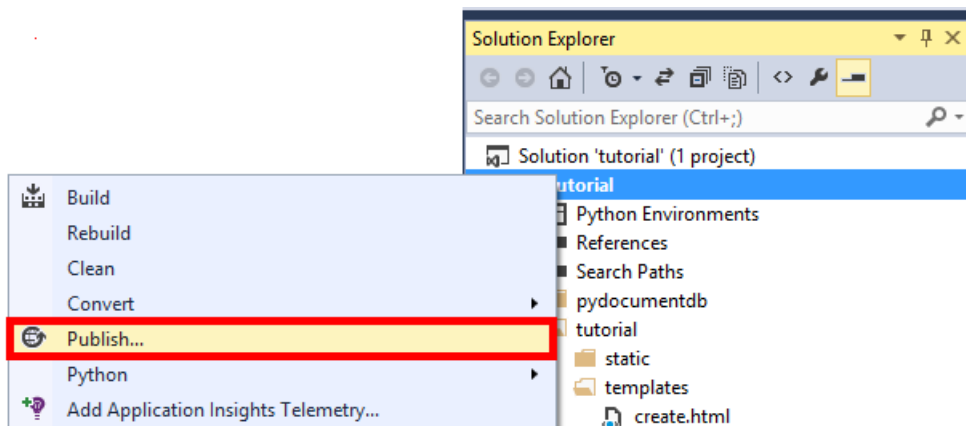


6. Stop debugging the project by pressing Shift+F5.

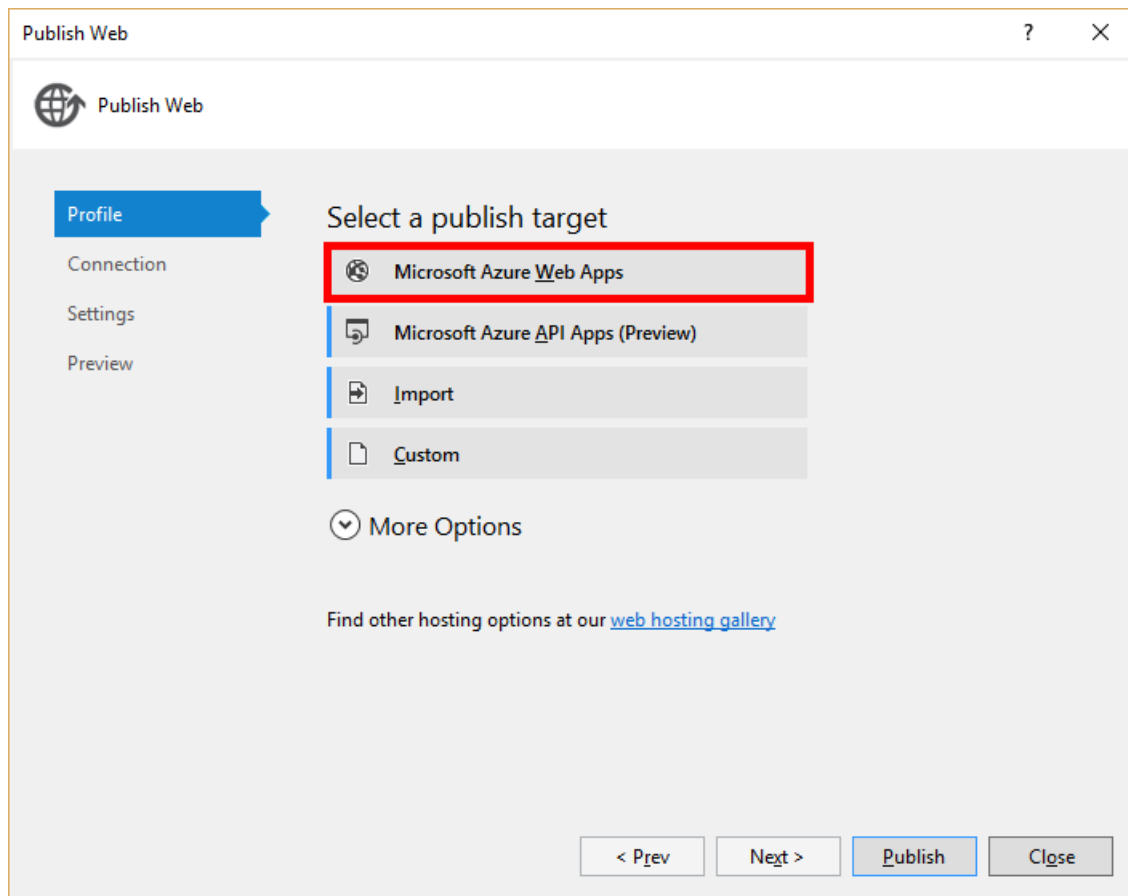
Step 5: Deploy the web application to Azure Websites

Now that you have the complete application working correctly against DocumentDB, we're going to deploy this to Azure Websites.

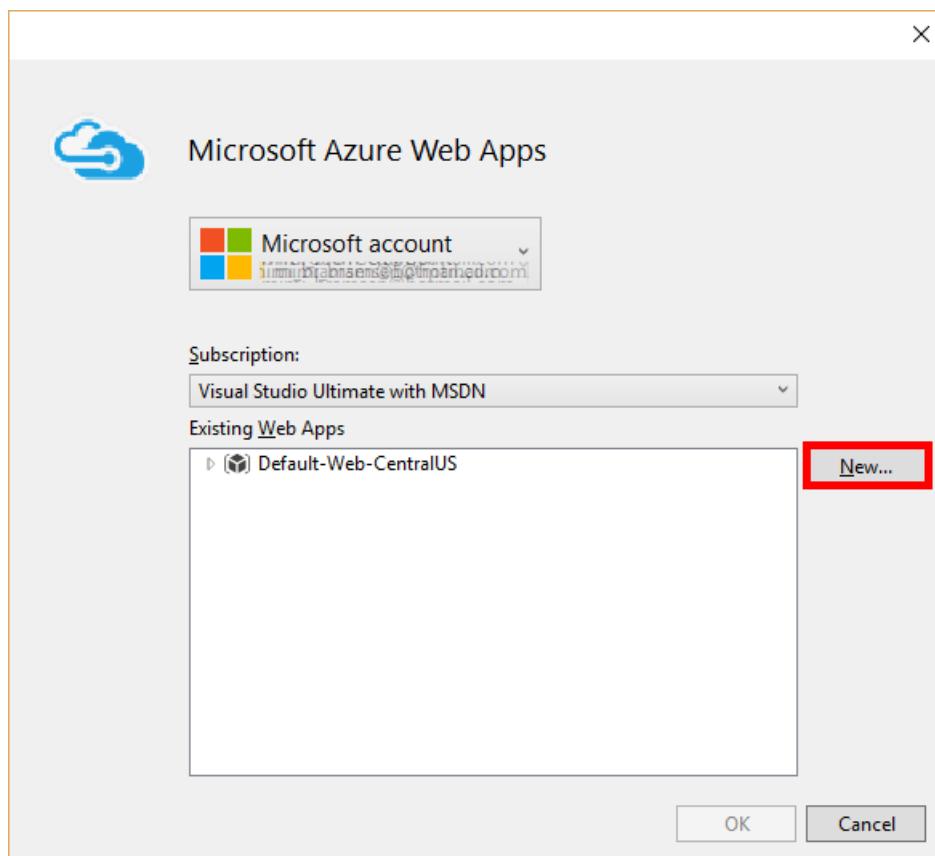
1. Right-click the project in Solution Explorer (make sure you're not still running it locally) and select **Publish**.



2. In the **Publish Web** window, select **Microsoft Azure Web Apps**, and then click **Next**.



3. In the **Microsoft Azure Web Apps Window** window, click **New**.



4. In the **Create site on Microsoft Azure** window, enter a **Web app name**, **App Service plan**, **Resource group**, and **Region**, then click **Create**.

Create Web App on Microsoft Azure

Create a Web App on Microsoft Azure

Microsoft account

Web App name: docdb-python-tutorial2 ✓
.azurewebsites.net

App Service plan: Default1 (Central US)

Resource group: Default-Web-CentralUS

Region: Central US

Database server: No database

If you have removed your spending limit or you are using Pay As You Go, there may be monetary impact if you provision additional resources. [legal terms](#)
[Learn more](#)

Create Cancel

5. In the **Publish Web** window, click **Publish**.

Publish Web

Profile

Connection

Settings

Preview

docdb-python-tutorial2

Publish method: Web Deploy

Server: docdb-python-tutorial2.scm.azurewebsites.net:443

Site name: docdb-python-tutorial2

User name: \$docdb-python-tutorial2

Password:

☒ Save password

Destination URL: http://docdb-python-tutorial2.azurewebsites.net

Validate Connection

< Prev Next > Publish Close

6. In a few seconds, Visual Studio will finish publishing your web application and launch a browser where you can see your handy work running in Azure!

Troubleshooting

If this is the first Python app you've run on your computer, ensure that the following folders (or the equivalent installation locations) are included in your PATH variable:

```
C:\Python27\site-packages;C:\Python27\;C:\Python27\Scripts;
```

If you receive an error on your vote page, and you named your project something other than **tutorial**, make sure that `__init__.py` references the correct project name in the line: `import tutorial.view`.

Next steps

Congratulations! You have just completed your first Python web application using Azure DocumentDB and published it to Azure Websites.

We update and improve this topic frequently based on your feedback. Once you've completed the tutorial, please using the voting buttons at the top and bottom of this page, and be sure to include your feedback on what improvements you want to see made. If you'd like us to contact you directly, feel free to include your email address in your comments.

To add additional functionality to your web application, review the APIs available in the [DocumentDB Python SDK](#).

For more information about Azure, Visual Studio, and Python, see the [Python Developer Center](#).

For additional Python Flask tutorials, see [The Flask Mega-Tutorial, Part I: Hello, World!](#).

Use the Azure DocumentDB Emulator for development and testing

11/22/2016 • 5 min to read • [Edit on GitHub](#)

Contributors

[arramac](#) • [mimig](#)

Download the Emulator

The Azure DocumentDB Emulator provides a local environment that emulates the Azure DocumentDB service for development purposes. Using the DocumentDB Emulator, you can develop and test your application locally, without creating an Azure subscription or incurring any costs. When you're satisfied with how your application is working in the DocumentDB Emulator, you can switch to using an Azure DocumentDB account in the cloud.

We recommend getting started by watching the following video, where Kirill Gavrylyuk shows how to get started with the DocumentDB Emulator.

DocumentDB Emulator system requirements

The DocumentDB Emulator has the following hardware and software requirements:

- Software requirements
 - Windows Server 2012 R2, Windows Server 2016, or Windows 10
- Minimum Hardware requirements
 - 2 GB RAM
 - 10 GB available hard disk space

Installing the DocumentDB Emulator

You can download and install the DocumentDB Emulator from the [Microsoft Download Center](#).

NOTE

To install, configure, and run the DocumentDB Emulator, you must have administrative privileges on the computer.

Checking for DocumentDB Emulator updates

The DocumentDB Emulator includes a built-in Azure DocumentDB Data Explorer to browse data stored within DocumentDB, create new collections, and let you know when a new update is available for download.

NOTE

Data created in one version of the DocumentDB Emulator is not guaranteed to be accessible when using a different version. If you need to persist your data for the long term, it is recommended that you store that data in an Azure DocumentDB account, rather than in the DocumentDB Emulator.

How the DocumentDB Emulator works

The DocumentDB Emulator provides a high-fidelity emulation of the DocumentDB service. It supports identical functionality as Azure DocumentDB, including support for creating and querying JSON documents, provisioning and scaling collections, and executing stored procedures and triggers. You can develop and test applications using the DocumentDB Emulator, and deploy them to Azure at global scale by just making a single configuration change to the connection endpoint for DocumentDB.

While we created a high-fidelity local emulation of the actual DocumentDB service, the implementation of the DocumentDB Emulator is different than that of the service. For example, the DocumentDB Emulator uses standard OS components such as the local file system for persistence, and HTTPS protocol stack for connectivity. This means that some functionality that relies on Azure infrastructure like global replication, single-digit millisecond latency for reads/writes, and tunable consistency levels are not available via the DocumentDB Emulator.

Authenticating requests against the DocumentDB Emulator

Just as with Azure Document in the cloud, every request that you make against the DocumentDB Emulator must be authenticated. The DocumentDB Emulator supports a single fixed account and a well-known authentication key for master key authentication. This account and key are the only credentials permitted for use with the DocumentDB Emulator. They are:

```
Account name: localhost:<port>
Account key: C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==
```

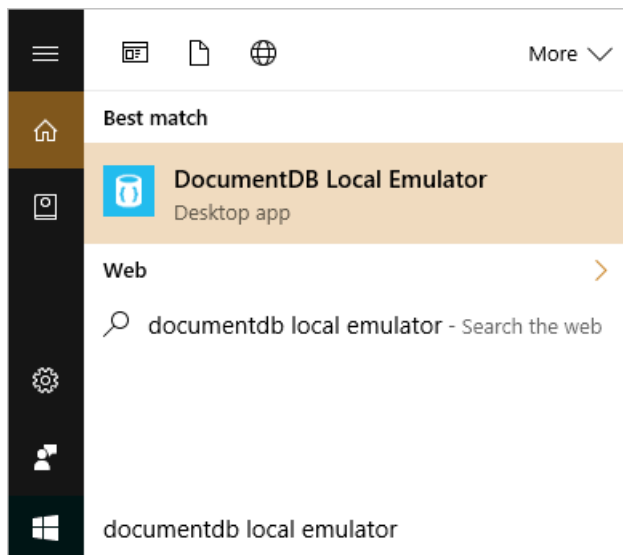
NOTE

The master key supported by the DocumentDB Emulator is intended for use only with the emulator. You cannot use your production DocumentDB account and key with the DocumentDB Emulator.

Additionally, just as the Azure DocumentDB service, the DocumentDB Emulator supports only secure communication via SSL.

Start and initialize the DocumentDB Emulator

To start the Azure DocumentDB Emulator, select the Start button or press the Windows key. Begin typing **DocumentDB Emulator**, and select the emulator from the list of applications.



When the emulator is running, you'll see an icon in the Windows taskbar notification area. The DocumentDB Emulator by default runs on the local machine ("localhost") listening on port 8081.



The DocumentDB Emulator is installed by default to the `C:\Program Files\Azure DocumentDB Emulator` directory. You can also start and stop the emulator from the command-line. See [command-line tool reference](#) for more information.

Developing with the DocumentDB Emulator

Once you have the DocumentDB Emulator running on your desktop, you can use any supported [DocumentDB SDK](#) or the [DocumentDB REST API](#) to interact with the Emulator. The DocumentDB Emulator also includes a built-in Data Explorer that lets you create collections, view and edit documents without writing any code.

```
// Connect to the DocumentDB Emulator running locally
DocumentClient client = new DocumentClient(
    new Uri("https://localhost:8081"),
    "C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==",
    new ConnectionPolicy { EnableEndpointDiscovery = false });
```

NOTE

When connecting to the emulator, you must set `EnableEndpointDiscovery = false` in the connection configuration.

If you're using [DocumentDB protocol support for MongoDB](#), please use the following connection string:

```
mongodb://localhost:C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==@localhost:10250/admin?ssl=true&tls.sslSelfSignedCerts=true
```

You can use existing tools like [DocumentDB Studio](#) to connect to the DocumentDB Emulator. You can also migrate data between the DocumentDB Emulator and the Azure DocumentDB service using the [DocumentDB Data Migration Tool](#).

DocumentDB Emulator command-line tool reference

From the installation location, you can use the command-line to start and stop the emulator, configure options, and perform other operations.

Command Line Syntax

```
DocumentDB.LocalEmulator.exe [/shutdown] [/datapath] [/port] [/mongoport] [/directports] [/key] [/?]
```

To view the list of options, type `DocumentDB.LocalEmulator.exe /?` at the command prompt.

Option	Description	Command	Arguments
[No arguments]	Starts up the DocumentDB Emulator with default settings	DocumentDB.LocalEmulator.exe	
Shutdown	Shuts down the DocumentDB Emulator	DocumentDB.LocalEmulator.exe /Shutdown	
Help	Displays the list of command line arguments	DocumentDB.LocalEmulator.exe /?	
Datapath	Specifies the path in which to store data files	DocumentDB.LocalEmulator.exe /datapath=<datapath>	<datapath>: An accessible path
Port	Specifies the port number to use for the emulator. Default is 8081	DocumentDB.LocalEmulator.exe /port=<port>	<port>: Single port number
MongoPort	Specifies the port number to use for MongoDB compatibility API. Default is 10250	DocumentDB.LocalEmulator.exe /mongoport=<mongoport>	<mongoport>: Single port number
DirectPorts	Specifies the ports to use for direct connectivity. Defaults are 10251,10252,10253,10254	DocumentDB.LocalEmulator.exe /directports:<directports>	<directports>: Comma delimited list of 4 ports
Key	Authorization key for the emulator. Key must be the base-64 encoding of a 64-byte vector	DocumentDB.LocalEmulator.exe /key:<key>	<key>: Key must be the base-64 encoding of a 64-byte vector
EnableThrottling	Specifies that request throttling behavior is enabled	DocumentDB.LocalEmulator.exe /enablethrottling	
DisableThrottling	Specifies that request throttling behavior is disabled	DocumentDB.LocalEmulator.exe /disablethrottling	

Differences between the DocumentDB Emulator and Azure DocumentDB

Because the DocumentDB Emulator provides an emulated environment running on a local developer workstation, there are some differences in functionality between the emulator and an Azure DocumentDB account in the cloud:

- The DocumentDB Emulator supports only a single fixed account and a well-known master key. Key

regeneration is not possible in the DocumentDB Emulator.

- The DocumentDB Emulator is not a scalable service and will not support a large number of collections.
- The DocumentDB Emulator does not simulate different [DocumentDB consistency levels](#).
- The DocumentDB Emulator does not simulate [multi-region replication](#).
- The DocumentDB Emulator does not support the service quota overrides that are available in the Azure DocumentDB service (e.g. document size limits, increased partitioned collection storage).
- As your copy of the DocumentDB Emulator might not be up to date with the most recent changes with the Azure DocumentDB service, please [DocumentDB capacity planner](#) to accurately estimate production throughput (RUs) needs of your application.

Next steps

- To learn more about DocumentDB, see [Introduction to Azure DocumentDB](#)
- To start developing against the DocumentDB Emulator, download one of the [supported DocumentDB SDKs](#).

Frequently asked questions about DocumentDB

11/22/2016 • 9 min to read • [Edit on GitHub](#)

Contributors

[mimig](#) • [Theano Petersen](#) • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [arramac](#) • [v-aljenk](#)

Database questions about Microsoft Azure DocumentDB fundamentals

What is Microsoft Azure DocumentDB?

Microsoft Azure DocumentDB is a blazing fast, planet-scale NoSQL document database-as-a-service that offers rich querying over schema-free data, helps deliver configurable and reliable performance, and enables rapid development, all through a managed platform backed by the power and reach of Microsoft Azure. DocumentDB is the right solution for web, mobile, gaming and IoT applications when predictable throughput, high availability, low latency, and a schema-free data model are key requirements. DocumentDB delivers schema flexibility and rich indexing via a native JSON data model, and includes multi-document transactional support with integrated JavaScript.

For more database questions, answers, and instructions on deploying and using this service, see the [DocumentDB documentation page](#).

What kind of database is DocumentDB?

DocumentDB is a NoSQL document oriented database that stores data in JSON format. DocumentDB supports nested, self-contained-data structures that can be queried through a rich DocumentDB [SQL query grammar](#). DocumentDB provides high-performance transactional processing of server-side JavaScript through [stored procedures, triggers, and user defined functions](#). The database also supports developer tunable consistency levels with associated [performance levels](#).

Do DocumentDB databases have tables like a relational database (RDBMS)?

No, DocumentDB stores data in collections of JSON documents. For information on DocumentDB resources, see [DocumentDB resource model and concepts](#). For more information about how NoSQL solutions such as DocumentDB differ from relational solutions, see [NoSQL vs SQL](#).

Do DocumentDB databases support schema-free data?

Yes, DocumentDB allows applications to store arbitrary JSON documents without schema definition or hints. Data is immediately available for query through the DocumentDB SQL query interface.

Does DocumentDB support ACID transactions?

Yes, DocumentDB supports cross-document transactions expressed as JavaScript stored procedures and triggers. Transactions are scoped to a single partition within each collection and executed with ACID semantics as all or nothing isolated from other concurrently executing code and user requests. If exceptions are thrown through the server-side execution of JavaScript application code, the entire transaction is rolled back. For more information about transactions, see [Database program transactions](#).

What are the typical use cases for DocumentDB?

DocumentDB is a good choice for new web, mobile, gaming and IoT applications where automatic scale, predictable performance, fast order of millisecond response times, and the ability to query over schema-free data is important. DocumentDB lends itself to rapid development and supporting the continuous iteration of application data models. Applications that manage user generated content and data are [common use cases for DocumentDB](#).

How does DocumentDB offer predictable performance?

A [request unit](#) is the measure of throughput in DocumentDB. 1 RU corresponds to the throughput of the GET of a 1KB document. Every operation in DocumentDB, including reads, writes, SQL queries, and stored procedure executions has a deterministic RU value based on the throughput required to complete the operation. Instead of thinking about CPU, IO and memory and how they each impact your application throughput, you can think in terms of a single RU measure.

Each DocumentDB collection can be reserved with provisioned throughput in terms of RUs of throughput per second. For applications of any scale, you can benchmark individual requests to measure their RU values, and provision collections to handle the sum total of request units across all requests. You can also scale up or scale down your collection's throughput as the needs of your application evolve. For more information about request units and for help determining your collection needs, please read [Manage Performance and Capacity](#) and try the [throughput calculator](#).

Is DocumentDB HIPAA compliant?

Yes, DocumentDB is HIPAA-compliant. HIPAA establishes requirements for the use, disclosure, and safeguarding of individually identifiable health information. For more information, see the [Microsoft Trust Center](#).

What are the storage limits of DocumentDB?

There is no limit to the total amount of data that a collection can store in DocumentDB. If you would like to store over 250 GB of data within a single collection, please [contact support](#) to have your account quota increased.

What are the throughput limits of DocumentDB?

There is no limit to the total amount of throughput that a collection can support in DocumentDB, if your workload can be distributed roughly evenly among a sufficiently large number of partition keys. If you wish to exceed 250,000 request units/second per collection or account, please [contact support](#) to have your account quota increased.

How much does Microsoft Azure DocumentDB cost?

Please refer to the [DocumentDB pricing details](#) page for details. DocumentDB usage charges are determined by the number of collections in use, the number of hours the collections were online, and the consumed storage and provisioned throughput for each collection.

Is there a free account available?

If you are new to Azure, you can sign up for an [Azure free account](#), which gives you 30 days and \$200 to try all the Azure services. Or, if you have a Visual Studio subscription, you are eligible for [\\$150 in free Azure credits per month](#) to use on any Azure service.

You can also use the [Azure DocumentDB Emulator](#) to develop and test your application locally for free, without creating an Azure subscription. When you're satisfied with how your application is working in the DocumentDB Emulator, you can switch to using an Azure DocumentDB account in the cloud.

How can I get additional help with DocumentDB?

If you need any help, please reach out to us on [Stack Overflow](#), the [Azure DocumentDB MSDN Developer Forums](#), or schedule a [1:1 chat with the DocumentDB engineering team](#). To stay up to date on the latest DocumentDB news and features, follow us on [Twitter](#).

Set up Microsoft Azure DocumentDB

How do I sign up for Microsoft Azure DocumentDB?

Microsoft Azure DocumentDB is available in the [Azure Portal](#). First you must sign up for a Microsoft Azure subscription. Once you sign up for a Microsoft Azure subscription, you can add a DocumentDB account to your Azure subscription. For instructions on adding a DocumentDB account, see [Create a DocumentDB database account](#).

What is a master key?

A master key is a security token to access all resources for an account. Individuals with the key have read and write access to the all resources in the database account. Use caution when distributing master keys. The primary master key and secondary master key are available in the **Keys** blade of the [Azure Portal](#). For more information about keys, see [View, copy, and regenerate access keys](#).

How do I create a database?

You can create databases using the [Azure Portal]() as described in [Create a DocumentDB database](#), one of the [DocumentDB SDKs](#), or through the [REST APIs](#).

What is a collection?

A collection is a container of JSON documents and the associated JavaScript application logic. A collection is a billable entity, where the [cost](#) is determined by the throughput and storage used. Collections can span one or more partitions/servers and can scale to handle practically unlimited volumes of storage or throughput.

Collections are also the billing entities for DocumentDB. Each collection is billed hourly based on the provisioned throughput and the storage space used. For more information, see [DocumentDB pricing](#).

How do I set up users and permissions?

You can create users and permissions using one of the [DocumentDB SDKs](#) or through the [REST APIs](#).

Database questions about developing against Microsoft Azure DocumentDB

How to do I start developing against DocumentDB?

[SDKs](#) are available for .NET, Python, Node.js, JavaScript, and Java. Developers can also leverage the [RESTful HTTP APIs](#) to interact with DocumentDB resources from a variety of platforms and languages.

Samples for the DocumentDB [.NET](#), [Java](#), [Node.js](#), and [Python](#) SDKs are available on GitHub.

Does DocumentDB support SQL?

The DocumentDB SQL query language is an enhanced subset of the query functionality supported by SQL. The DocumentDB SQL query language provides rich hierarchical and relational operators and extensibility via JavaScript based user-defined functions (UDFs). JSON grammar allows for modeling JSON documents as trees with labels as the tree nodes, which is used by both the DocumentDB automatic indexing techniques as well as the SQL query dialect of DocumentDB. For details on how to use the SQL grammar, please see the [Query DocumentDB](#) article.

What are the data types supported by DocumentDB?

The primitive data types supported in DocumentDB are the same as JSON. JSON has a simple type system that consists of Strings, Numbers (IEEE754 double precision), and Booleans - true, false, and Nulls. More complex data types like DateTime, Guid, Int64, and Geometry can be represented both in JSON and DocumentDB through the creation of nested objects using the { } operator and arrays using the [] operator.

How does DocumentDB provide concurrency?

DocumentDB supports optimistic concurrency control (OCC) through HTTP entity tags or etags. Every DocumentDB resource has an etag, and the etag is set on the server every time a document is updated. The etag header and the current value are included in all response messages. Etags can be used with the If-Match header to allow the server to decide if a resource should be updated. The If-Match value is the etag value to be checked against. If the etag value matches the server etag value, the resource will be updated. If the etag is no longer current, the server rejects the operation with an "HTTP 412 Precondition failure" response code. The client will then have to refetch the resource to acquire the current etag value for the resource. In addition, etags can be used with If-None-Match header to determine if a re-fetch of a resource is needed.

To use optimistic concurrency in .NET, use the [AccessCondition](#) class. For a .NET sample, see [Program.cs](#) in the

DocumentManagement sample on github.

How do I perform transactions in DocumentDB?

DocumentDB supports language-integrated transactions via JavaScript stored procedures and triggers. All database operations inside scripts are executed under snapshot isolation scoped to the collection if it is a single-partition collection, or documents with the same partition key value within a collection, if the collection is partitioned. A snapshot of the document versions (ETags) is taken at the start of the transaction and committed only if the script succeeds. If the JavaScript throws an error, the transaction is rolled back. See [DocumentDB server-side programming](#) for more details.

How can I bulk insert documents into DocumentDB?

There are three ways to bulk insert documents into DocumentDB:

- The data migration tool, as described in [Import data to DocumentDB](#).
- Document Explorer in the Azure Portal, as described in [Bulk add documents with Document Explorer](#).
- Stored procedures, as described in [DocumentDB server-side programming](#).

Does DocumentDB support resource link caching?

Yes, because DocumentDB is a RESTful service, resource links are immutable and can be cached. DocumentDB clients can specify an "If-None-Match" header for reads against any resource like document or collection and update their local copies only when the server version has change.

Is a local instance of DocumentDB available?

Yes. The [Azure DocumentDB Emulator](#) provides a high-fidelity emulation of the DocumentDB service. It supports identical functionality as Azure DocumentDB, including support for creating and querying JSON documents, provisioning and scaling collections, and executing stored procedures and triggers. You can develop and test applications using the DocumentDB Emulator, and deploy them to Azure at global scale by just making a single configuration change to the connection endpoint for DocumentDB.

Storage and predictable performance provisioning in DocumentDB

11/15/2016 • 8 min to read • [Edit on GitHub](#)

Contributors

Syam Nair • mimig • Theano Petersen • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • arramac • Stephen Baron • v-aljenk

Azure DocumentDB is a fully managed, scalable document-oriented NoSQL database service for JSON documents. With DocumentDB, you don't have to rent virtual machines, deploy software, or monitor databases. DocumentDB is operated and continuously monitored by Microsoft engineers to deliver world class availability, performance, and data protection.

You can get started with DocumentDB by [creating a database account](#) and a [DocumentDB database](#) through the [Azure portal](#). DocumentDB databases are offered in units of solid-state drive (SSD) backed storage and throughput. These storage units are provisioned by [creating database collections](#) within your database account, each collection with reserved throughput that can be scaled up or down at any time to meet the demands of your application.

If your application exceeds your reserved throughput for one or multiple collections, requests are limited on a per collection basis. This means that some application requests may succeed while others may be throttled.

This article provides an overview of the resources and metrics available to manage capacity and plan data storage.

Database account

As an Azure subscriber, you can provision one or more DocumentDB database accounts to manage your database resources. Each subscription is associated with a single Azure subscription.

DocumentDB accounts can be created through the [Azure portal](#), or by using [an ARM template](#) or [Azure CLI](#).

Databases

A single DocumentDB database can contain practically an unlimited amount of document storage grouped into collections. Collections provide performance isolation - each collection can be provisioned with throughput that is not shared with other collections in the same database or account. A DocumentDB database is elastic in size, ranging from GBs to TBs of SSD backed document storage and provisioned throughput. Unlike a traditional RDBMS database, a database in DocumentDB is not scoped to a single machine and can span multiple machines or clusters.

With DocumentDB, as you need to scale your applications, you can create more collections or databases or both. Databases can be created through the [Azure portal](#) or through any one of the [DocumentDB SDKs](#).

Database collections

Each DocumentDB database can contain one or more collections. Collections act as highly available data partitions for document storage and processing. Each collection can store documents with heterogeneous schema.

DocumentDB's automatic indexing and query capabilities allow you to easily filter and retrieve documents. A collection provides the scope for document storage and query execution. A collection is also a transaction domain for all the documents contained within it. Collections are allocated throughput based on the value set in the Azure

portal or via the SDKs.

Collections are automatically partitioned into one or more physical servers by DocumentDB. When you create a collection, you can specify the provisioned throughput in terms of request units per second and a partition key property. The value of this property is used by DocumentDB to distribute documents among partitions and route requests like queries. The partition key value also acts as the transaction boundary for stored procedures and triggers. Each collection has a reserved amount of throughput specific to that collection, which is not shared with other collections in the same account. Therefore, you can scale out your application both in terms of storage and throughput.

Collections can be created through the [Azure portal](#) or through any one of the [DocumentDB SDKs](#).

Request units and database operations

When you create a collection, you reserve throughput in terms of [request units \(RU\)](#) per second. Instead of thinking about and managing hardware resources, you can think of a **request unit** as a single measure for the resources required to perform various database operations and service an application request. A read of a 1 KB document consumes the same 1 RU regardless of the number of items stored in the collection or the number of concurrent requests running at the same. All requests against DocumentDB, including complex operations like SQL queries have a predictable RU value that can be determined at development time. If you know the size of your documents and the frequency of each operation (reads, writes and queries) to support for your application, you can provision the exact amount of throughput to meet your application's needs, and scale your database up and down as your performance needs change.

Each collection can be reserved with throughput in blocks of 100 request units per second, from hundreds up to millions of request units per second. The provisioned throughput can be adjusted throughout the life of a collection to adapt to the changing processing needs and access patterns of your application. For more information, see [DocumentDB performance levels](#).

IMPORTANT

Collections are billable entities. The cost is determined by the provisioned throughput of the collection measured in request units per second along with the total consumed storage in gigabytes.

How many request units will a particular operation like insert, delete, query, or stored procedure execution consume? A request unit is a normalized measure of request processing cost. A read of a 1 KB document is 1 RU, but a request to insert, replace or delete the same document will consume more processing from the service and thereby more request units. Each response from the service includes a custom header (`x-ms-request-charge`) that reports the request units consumed for the request. This header is also accessible through the [SDKs](#). In the .NET SDK, [RequestCharge](#) is a property of the [ResourceResponse](#) object. If you want to estimate your throughput needs before making a single call, you can use the [capacity planner](#) to help with this estimation.

NOTE

The baseline of 1 request unit for a 1 KB document corresponds to a simple GET of the document with [Session Consistency](#).

There are several factors that impact the request units consumed for an operation against a DocumentDB database account. These factors include:

- Document size. As document sizes increase the units consumed to read or write the data will also increase.
- Property count. Assuming default indexing of all properties, the units consumed to write a document will increase as the property count increases.
- Data consistency. When using data consistency levels of Strong or Bounded Staleness, additional units will be consumed to read documents.

- Indexed properties. An index policy on each collection determines which properties are indexed by default. You can reduce your request unit consumption by limiting the number of indexed properties.
- Document indexing. By default each document is automatically indexed, you will consume fewer request units if you choose not to index some of your documents.

For more information, see [DocumentDB request units](#).

For example, here's a table that shows how many request units to provision at three different document sizes (1KB, 4KB, and 64KB) and at two different performance levels (500 reads/second + 100 writes/second and 500 reads/second + 500 writes/second). The data consistency was configured at Session, and the indexing policy was set to None.

Document size	Reads/second	Writes/second	Request units
1 KB	500	100	$(500 * 1) + (100 * 5) = 1,000 \text{ RU/s}$
1 KB	500	500	$(500 * 5) + (100 * 5) = 3,000 \text{ RU/s}$
4 KB	500	100	$(500 * 1.3) + (100 * 7) = 1,350 \text{ RU/s}$
4 KB	500	500	$(500 * 1.3) + (500 * 7) = 4,150 \text{ RU/s}$
64 KB	500	100	$(500 * 10) + (100 * 48) = 9,800 \text{ RU/s}$
64 KB	500	500	$(500 * 10) + (500 * 48) = 29,000 \text{ RU/s}$

Queries, stored procedures, and triggers consume request units based on the complexity of the operations being performed. As you develop your application, inspect the request charge header to better understand how each operation is consuming request unit capacity.

Choice of consistency level and throughput

The choice of default consistency level has an impact on the throughput and latency. You can set the default consistency level both programmatically and through the Azure portal. You can also override the consistency level on a per request basis. By default, the consistency level is set to **Session**, which provides monotonic read/writes and read your write guarantees. Session consistency is great for user-centric applications and provides an ideal balance of consistency and performance trade-offs.

For instructions on changing your consistency level on the Azure portal, see [How to Manage a DocumentDB Account](#). Or, for more information on consistency levels, see [Using consistency levels](#).

Provisioned document storage and index overhead

DocumentDB supports the creation of both single-partition and partitioned collections. Each partition in

DocumentDB supports up to 10 GB of SSD backed storage. The 10GB of document storage includes the documents plus storage for the index. By default, a DocumentDB collection is configured to automatically index all of the documents without explicitly requiring any secondary indices or schema. Based on applications using DocumentDB, the typical index overhead is between 2-20%. The indexing technology used by DocumentDB ensures that regardless of the values of the properties, the index overhead does not exceed more than 80% of the size of the documents with default settings.

By default all documents are indexed by DocumentDB automatically. However, if you want to fine-tune the index overhead, you can choose to remove certain documents from being indexed at the time of inserting or replacing a document, as described in [DocumentDB indexing policies](#). You can configure a DocumentDB collection to exclude all documents within the collection from being indexed. You can also configure a DocumentDB collection to selectively index only certain properties or paths with wildcards of your JSON documents, as described in [Configuring the indexing policy of a collection](#). Excluding properties or documents also improves the write throughput – which means you will consume fewer request units.

Next steps

To continue learning about how DocumentDB works, see [Partitioning and scaling in Azure DocumentDB](#).

For instructions on monitoring performance levels on the Azure portal, see [Monitor a DocumentDB account](#). For more information on choosing performance levels for collections, see [Performance levels in DocumentDB](#).

Partitioning and scaling in Azure DocumentDB

11/15/2016 • 16 min to read • [Edit on GitHub](#)

Contributors

arramac • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [mimig](#) • [Rajesh Nagpal](#) • [Andrew Hoh](#) • [jayantacs](#)
• [Ross McAllister](#) • [John Macintyre](#)

[Microsoft Azure DocumentDB](#) is designed to help you achieve fast, predictable performance and scale seamlessly along with your application as it grows. This article provides an overview of how partitioning works in DocumentDB, and describes how you can configure DocumentDB collections to effectively scale your applications.

After reading this article, you will be able to answer the following questions:

- How does partitioning work in Azure DocumentDB?
- How do I configure partitioning in DocumentDB
- What are partition keys, and how do I pick the right partition key for my application?

To get started with code, please download the project from [DocumentDB Performance Testing Driver Sample](#).

Partitioning in DocumentDB

In DocumentDB, you can store and query schema-less JSON documents with order-of-millisecond response times at any scale. DocumentDB provides containers for storing data called **collections**. Collections are logical resources and can span one or more physical partitions or servers. The number of partitions is determined by DocumentDB based on the storage size and the provisioned throughput of the collection. Every partition in DocumentDB has a fixed amount of SSD-backed storage associated with it, and is replicated for high availability. Partition management is fully managed by Azure DocumentDB, and you do not have to write complex code or manage your partitions. DocumentDB collections are **practically unlimited** in terms of storage and throughput.

Partitioning is completely transparent to your application. DocumentDB supports fast reads and writes, SQL and LINQ queries, JavaScript based transactional logic, consistency levels, and fine-grained access control via REST API calls to a single collection resource. The service handles distributing data across partitions and routing query requests to the right partition.

How does this work? When you create a collection in DocumentDB, you'll notice that there's a **partition key property** configuration value that you can specify. This is the JSON property (or path) within your documents that can be used by DocumentDB to distribute your data among multiple servers or partitions. DocumentDB will hash the partition key value and use the hashed result to determine the partition in which the JSON document will be stored. All documents with the same partition key will be stored in the same partition.

For example, consider an application that stores data about employees and their departments in DocumentDB.

Let's choose `"department"` as the partition key property, in order to scale out data by department. Every document in DocumentDB must contain a mandatory `"id"` property that must be unique for every document with the same partition key value, e.g. `"Marketing"`. Every document stored in a collection must have a unique combination of partition key and id, e.g. `{ "Department": "Marketing", "id": "0001" }`,

`{ "Department": "Marketing", "id": "0002" }`, and `{ "Department": "Sales", "id": "0001" }`.

In other words, the compound property of (partition key, id) is the primary key for your collection.

Partition keys

The choice of the partition key is an important decision that you'll have to make at design time. You must pick a JSON property name that has a wide range of values and is likely to have evenly distributed access patterns. The partition key is specified as a JSON path, e.g. `/department` represents the property department.

The following table shows examples of partition key definitions and the JSON values corresponding to each.

Partition Key Path	Description
<code>/department</code>	Corresponds to the JSON value of <code>doc.department</code> where <code>doc</code> is the document.
<code>/properties/name</code>	Corresponds to the JSON value of <code>doc.properties.name</code> where <code>doc</code> is the document (nested property).
<code>/id</code>	Corresponds to the JSON value of <code>doc.id</code> (<code>id</code> and partition key are the same property).
<code>/"department name"</code>	Corresponds to the JSON value of <code>doc["department name"]</code> where <code>doc</code> is the document.

NOTE

The syntax for partition key path is similar to the path specification for indexing policy paths with the key difference that the path corresponds to the property instead of the value, i.e. there is no wild card at the end. For example, you would specify `/department/?` to index the values under department, but specify `/department` as the partition key definition. The partition key path is implicitly indexed and cannot be excluded from indexing using indexing policy overrides.

Let's take a look at how the choice of partition key impacts the performance of your application.

Partitioning and provisioned throughput

DocumentDB is designed for predictable performance. When you create a collection, you reserve throughput in terms of **request units (RU) per second**. Each request is assigned a request unit charge that is proportionate to the amount of system resources like CPU and IO consumed by the operation. A read of a 1 kB document with Session consistency consumes 1 request unit. A read is 1 RU regardless of the number of items stored or the number of concurrent requests running at the same. Larger documents require higher request units depending on the size. If you know the size of your entities and the number of reads you need to support for your application, you can provision the exact amount of throughput required for your application's read needs.

When DocumentDB stores documents, it distributes them evenly among partitions based on the partition key value. The throughput is also distributed evenly among the available partitions i.e. the throughput per partition = (total throughput per collection)/ (number of partitions).

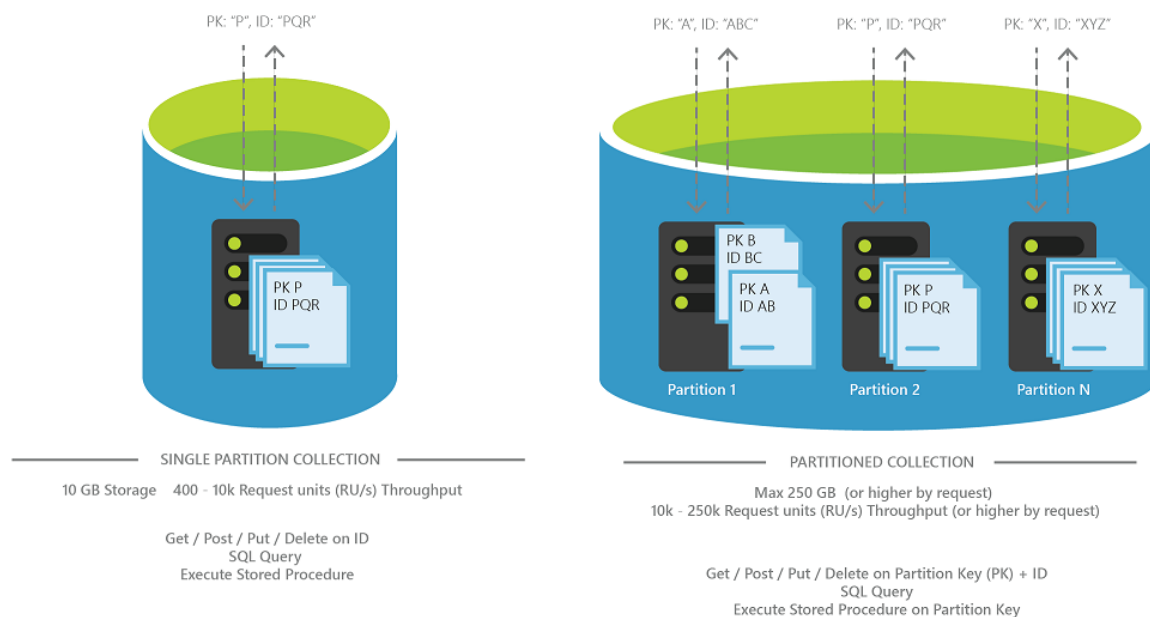
NOTE

In order to achieve the full throughput of the collection, you must choose a partition key that allows you to evenly distribute requests among a number of distinct partition key values.

Single Partition and Partitioned Collections

DocumentDB supports the creation of both single-partition and partitioned collections.

- **Partitioned collections** can span multiple partitions and support very large amounts of storage and throughput. You must specify a partition key for the collection.
- **Single-partition collections** have lower price options and the ability to query and perform transactions across all collection data. They have the scalability and storage limits of a single partition. You do not have to specify a partition key for these collections.



For scenarios that do not need large volumes of storage or throughput, single partition collections are a good fit. Note that single-partition collections have the scalability and storage limits of a single partition, i.e. up to 10 GB of storage and up to 10,000 request units per second.

Partitioned collections can support very large amounts of storage and throughput. The default offers however are configured to store up to 250 GB of storage and scale up to 250,000 request units per second. If you need higher storage or throughput per collection, please contact [Azure Support](#) to have these increased for your account.

The following table lists differences in working with a single-partition and partitioned collections:

	Single Partition Collection	Partitioned Collection
Partition Key	None	Required
Primary Key for Document	"id"	compound key <partition key> and "id"
Minimum Storage	0 GB	0 GB
Maximum Storage	10 GB	Unlimited (250 GB by default)
Minimum Throughput	400 request units per second	10,000 request units per second
Maximum Throughput	10,000 request units per second	Unlimited (250,000 request units per second by default)

API versions	All	API 2015-12-16 and newer
--------------	-----	--------------------------

Working with the SDKs

Azure DocumentDB added support for automatic partitioning with [REST API version 2015-12-16](#). In order to create partitioned collections, you must download SDK versions 1.6.0 or newer in one of the supported SDK platforms (.NET, Node.js, Java, Python).

Creating partitioned collections

The following sample shows a .NET snippet to create a collection to store device telemetry data of 20,000 request units per second of throughput. The SDK sets the OfferThroughput value (which in turn sets the `x-ms-offer-throughput` request header in the REST API). Here we set the `/deviceId` as the partition key. The choice of partition key is saved along with the rest of the collection metadata like name and indexing policy.

For this sample, we picked `deviceId` since we know that (a) since there are a large number of devices, writes can be distributed across partitions evenly and allowing us to scale the database to ingest massive volumes of data and (b) many of the requests like fetching the latest reading for a device are scoped to a single `deviceId` and can be retrieved from a single partition.

```
DocumentClient client = new DocumentClient(new Uri(endpoint), authKey);
await client.CreateDatabaseAsync(new Database { Id = "db" });

// Collection for device telemetry. Here the JSON property deviceId will be used as the partition key to
// spread across partitions. Configured for 10K RU/s throughput and an indexing policy that supports
// sorting against any number or string property.
DocumentCollection myCollection = new DocumentCollection();
myCollection.Id = "coll";
myCollection.PartitionKey.Paths.Add("/deviceId");

await client.CreateDocumentCollectionAsync(
    UriFactory.CreateDatabaseUri("db"),
    myCollection,
    new RequestOptions { OfferThroughput = 20000 });
```

NOTE

In order to create partitioned collections, you must specify a throughput value of > 10,000 request units per second. Since throughput is in multiples of 100, this has to be 10,100 or higher.

This method makes a REST API call to DocumentDB, and the service will provision a number of partitions based on the requested throughput. You can change the throughput of a collection as your performance needs evolve. See [Performance Levels](#) for more details.

Reading and writing documents

Now, let's insert data into DocumentDB. Here's a sample class containing a device reading, and a call to `CreateDocumentAsync` to insert a new device reading into a collection.

```

public class DeviceReading
{
    [JsonProperty("id")]
    public string Id;

    [JsonProperty("deviceId")]
    public string DeviceId;

    [JsonConverter(typeof(IsoDateTimeConverter))]
    [JsonProperty("readingTime")]
    public DateTime ReadingTime;

    [JsonProperty("metricType")]
    public string MetricType;

    [JsonProperty("unit")]
    public string Unit;

    [JsonProperty("metricValue")]
    public double MetricValue;
}

// Create a document. Here the partition key is extracted as "XMS-0001" based on the collection definition
await client.CreateDocumentAsync(
    UriFactory.CreateDocumentCollectionUri("db", "coll"),
    new DeviceReading
    {
        Id = "XMS-001-FE24C",
        DeviceId = "XMS-0001",
        MetricType = "Temperature",
        MetricValue = 105.00,
        Unit = "Fahrenheit",
        ReadingTime = DateTime.UtcNow
    });

```

Let's read the document by its partition key and id, update it, and then as a final step, delete it by partition key and id. Note that the reads include a PartitionKey value (corresponding to the `x-ms-documentdb-partitionkey` request header in the REST API).

```

// Read document. Needs the partition key and the ID to be specified
Document result = await client.ReadDocumentAsync(
    UriFactory.CreateDocumentUri("db", "coll", "XMS-001-FE24C"),
    new RequestOptions { PartitionKey = new PartitionKey("XMS-0001") });

DeviceReading reading = (DeviceReading)(dynamic)result;

// Update the document. Partition key is not required, again extracted from the document
reading.MetricValue = 104;
reading.ReadingTime = DateTime.UtcNow;

await client.ReplaceDocumentAsync(
    UriFactory.CreateDocumentUri("db", "coll", "XMS-001-FE24C"),
    reading);

// Delete document. Needs partition key
await client.DeleteDocumentAsync(
    UriFactory.CreateDocumentUri("db", "coll", "XMS-001-FE24C"),
    new RequestOptions { PartitionKey = new PartitionKey("XMS-0001") });

```

Querying partitioned collections

When you query data in partitioned collections, DocumentDB automatically routes the query to the partitions corresponding to the partition key values specified in the filter (if there are any). For example, this query is routed to just the partition containing the partition key "XMS-0001".

```
// Query using partition key
IQueryable<DeviceReading> query = client.CreateDocumentQuery<DeviceReading>(
    UriFactory.CreateDocumentCollectionUri("db", "coll"))
    .Where(m => m.MetricType == "Temperature" && m.DeviceId == "XMS-0001");
```

The following query does not have a filter on the partition key (DeviceId) and is fanned out to all partitions where it is executed against the partition's index. Note that you have to specify the `EnableCrossPartitionQuery` (`x-ms-documentdb-query-enablecrosspartition` in the REST API) to have the SDK to execute a query across partitions.

```
// Query across partition keys
IQueryable<DeviceReading> crossPartitionQuery = client.CreateDocumentQuery<DeviceReading>(
    UriFactory.CreateDocumentCollectionUri("db", "coll"),
    new FeedOptions { EnableCrossPartitionQuery = true })
    .Where(m => m.MetricType == "Temperature" && m.MetricValue > 100);
```

Parallel Query Execution

The DocumentDB SDKs 1.9.0 and above support parallel query execution options, which allow you to perform low latency queries against partitioned collections, even when they need to touch a large number of partitions. For example, the following query is configured to run in parallel across partitions.

```
// Cross-partition Order By Queries
IQueryable<DeviceReading> crossPartitionQuery = client.CreateDocumentQuery<DeviceReading>(
    UriFactory.CreateDocumentCollectionUri("db", "coll"),
    new FeedOptions { EnableCrossPartitionQuery = true, MaxDegreeOfParallelism = 10, MaxBufferedItemCount = 100})
    .Where(m => m.MetricType == "Temperature" && m.MetricValue > 100)
    .OrderBy(m => m.MetricValue);
```

You can manage parallel query execution by tuning the following parameters:

- By setting `MaxDegreeOfParallelism`, you can control the degree of parallelism i.e., the maximum number of simultaneous network connections to the collection's partitions. If you set this to -1, the degree of parallelism is managed by the SDK. If the `MaxDegreeOfParallelism` is not specified or set to 0, which is the default value, there will be a single network connection to the collection's partitions.
- By setting `MaxBufferedItemCount`, you can trade off query latency and client side memory utilization. If you omit this parameter or set this to -1, the number of items buffered during parallel query execution is managed by the SDK.

Given the same state of the collection, a parallel query will return results in the same order as in serial execution. When performing a cross-partition query that includes sorting (ORDER BY and/or TOP), the DocumentDB SDK issues the query in parallel across partitions and merges partially sorted results in the client side to produce globally ordered results.

Executing stored procedures

You can also execute atomic transactions against documents with the same device ID, e.g. if you're maintaining aggregates or the latest state of a device in a single document.

```
await client.ExecuteStoredProcedureAsync<DeviceReading>(
    UriFactory.CreateStoredProcedureUri("db", "coll", "SetLatestStateAcrossReadings"),
    new RequestOptions { PartitionKey = new PartitionKey("XMS-001") },
    "XMS-001-FE24C");
```

In the next section, we look at how you can move to partitioned collections from single-partition collections.

Migrating from single-partition to partitioned collections

When an application using a single-partition collection needs higher throughput (> 10,000 RU/s) or larger data storage (> 10GB), you can use the [DocumentDB Data Migration Tool](#) to migrate the data from the single-partition collection to a partitioned collection.

To migrate from a single-partition collection to a partitioned collection

1. Export data from the single-partition collection to JSON. See [Export to JSON file](#) for additional details.
2. Import the data into a partitioned collection created with a partition key definition and over 10,000 request units per second throughput, as shown in the example below. See [Import to DocumentDB](#) for additional details.

The screenshot shows the 'DocumentDB Data Migration Tool' window with the 'Target Information' tab selected. The 'Specify target information' section contains the following fields:

- Export to:** DocumentDB - Sequential record import (partitioned collection)
- Connection String:** DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database> (with a 'Verify' button)
- Collection:** newCollection
- Partition Key:** /deviceid
- Collection Throughput:** 20000
- Id Field:** empid
- Advanced Options:**
 - Number of Parallel Requests:** 100
 - ☐ Disable Automatic Id Generation
 - ☐ Update Existing Documents
 - Persist Date and Time as:** String
 - ☒ Enter Indexing Policy ☐ Select Policy File

At the bottom right, there are 'Previous' and 'Next' buttons.

TIP

For faster import times, consider increasing the Number of Parallel Requests to 100 or higher to take advantage of the higher throughput available for partitioned collections.

Now that we've completed the basics, let's look at a few important design considerations when working with partition keys in DocumentDB.

Designing for partitioning

The choice of the partition key is an important decision that you'll have to make at design time. This section describes some of the tradeoffs involved in selecting a partition key for your collection.

Partition key as the transaction boundary

Your choice of partition key should balance the need to enable the use of transactions against the requirement to distribute your entities across multiple partition keys to ensure a scalable solution. At one extreme, you could set the same partition key for all your documents, but this may limit the scalability of your solution. At the other extreme, you could assign a unique partition key for each document, which would be highly scalable but would prevent you from using cross document transactions via stored procedures and triggers. An ideal partition key is one that enables you to use efficient queries and that has sufficient cardinality to ensure your solution is scalable.

Avoiding storage and performance bottlenecks

It is also important to pick a property which allows writes to be distributed across a number of distinct values. Requests to the same partition key cannot exceed the throughput of a single partition, and will be throttled. So it is important to pick a partition key that does not result in **"hot spots"** within your application. The total storage size for documents with the same partition key can also not exceed 10 GB in storage.

Examples of good partition keys

Here are a few examples for how to pick the partition key for your application:

- If you're implementing a user profile backend, then the user ID is a good choice for partition key.
- If you're storing IoT data e.g. device state, a device ID is a good choice for partition key.
- If you're using DocumentDB for logging time-series data, then the hostname or process ID is a good choice for partition key.
- If you have a multi-tenant architecture, the tenant ID is a good choice for partition key.

Note that in some use cases (like the IoT and user profiles described above), the partition key might be the same as your id (document key). In others like the time series data, you might have a partition key that's different than the id.

Partitioning and logging/time-series data

One of the most common use cases of DocumentDB is for logging and telemetry. It is important to pick a good partition key since you might need to read/write vast volumes of data. The choice will depend on your read and write rates and kinds of queries you expect to run. Here are some tips on how to choose a good partition key.

- If your use case involves a small rate of writes accumulating over a long period of time, and need to query by ranges of timestamps and other filters, then using a rolup of the timestamp e.g. date as a partition key is a good approach. This allows you to query over all the data for a date from a single partition.
- If your workload is write heavy, which is generally more common, you should use a partition key that's not based on timestamp so that DocumentDB can distribute writes evenly across a number of partitions. Here a hostname, process ID, activity ID, or another property with high cardinality is a good choice.
- A third approach is a hybrid one where you have multiple collections, one for each day/month and the partition key is a granular property like hostname. This has the benefit that you can set different performance levels based on the time window, e.g. the collection for the current month is provisioned with higher throughput since it serves reads and writes, whereas previous months with lower throughput since they only serve reads.

Partitioning and multi-tenancy

If you are implementing a multi-tenant application using DocumentDB, there are two major patterns for implementing tenancy with DocumentDB – one partition key per tenant, and one collection per tenant. Here are the pros and cons for each:

- One Partition Key per tenant: In this model, tenants are colocated within a single collection. But queries and inserts for documents within a single tenant can be performed against a single partition. You can also implement transactional logic across all documents within a tenant. Since multiple tenants share a collection, you can save storage and throughput costs by pooling resources for tenants within a single collection rather than provisioning extra headroom for each tenant. The drawback is that you do not have performance

isolation per tenant. Performance/throughput increases apply to the entire collection vs targeted increases for tenants.

- One Collection per tenant: Each tenant has its own collection. In this model, you can reserve performance per tenant. With DocumentDB's new consumption based pricing model, this model is more cost-effective for multi-tenant applications with a small number of tenants.

You can also use a combination/tiered approach that collocates small tenants and migrates larger tenants to their own collection.

Next Steps

In this article, we've described how partitioning works in Azure DocumentDB, how you can create partitioned collections, and how you can pick a good partition key for your application.

- Perform scale and performance testing with DocumentDB. See [Performance and Scale Testing with Azure DocumentDB](#) for a sample.
- Get started coding with the [SDKs](#) or the [REST API](#)
- Learn about [provisioned throughput in DocumentDB](#)
- If you would like to customize how your application performs partitioning, you can plug in your own client-side partitioning implementation. See [Client-side partitioning support](#).

Consistency levels in DocumentDB

11/22/2016 • 6 min to read • [Edit on GitHub](#)

Contributors

Syam Nair • [mimig](#) • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Digvijay Makwana • Kirat Pandya • [v-aljenk](#) • [John Macintyre](#)

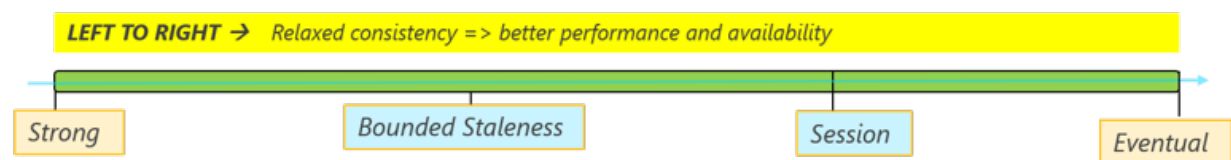
Azure DocumentDB is designed from the ground up with global distribution in mind. It is designed to offer predictable low latency guarantees, a 99.99% availability SLA, and multiple well-defined relaxed consistency models. Currently, DocumentDB provides four consistency levels: strong, bounded-staleness, session, and eventual. Besides the **strong** and the **eventual consistency** models commonly offered by other NoSQL databases, DocumentDB also offers two carefully codified and operationalized consistency models – **bounded staleness** and **session**, and has validated their usefulness against real world use cases. Collectively these four consistency levels enable you to make well-reasoned trade-offs between consistency, availability, and latency.

Scope of consistency

The granularity of consistency is scoped to a single user request. A write request may correspond to an insert, replace, upsert, or delete transaction (with or without the execution of an associated pre or post trigger). Or a write request may correspond to the transactional execution of a JavaScript stored procedure operating over multiple documents within a partition. As with the writes, a read/query transaction is also scoped to a single user request. The user may be required to paginate over a large result-set, spanning multiple partitions, but each read transaction is scoped to a single page and served from within a single partition.

Consistency levels

You can configure a default consistency level on your database account that applies to all the collections (across all of the databases) under your database account. By default, all reads and queries issued against the user defined resources will use the default consistency level specified on the database account. However, you can relax the consistency level of a specific read/query request by specifying the [\[x-ms-consistency-level\]](#) request header. There are four types of consistency levels supported by the DocumentDB replication protocol that provide a clear trade-off between specific consistency guarantees and performance, as described below.



Strong:

- Strong consistency offers a [linearizability](#) guarantee with the reads guaranteed to return the most recent version of a document.
- Strong consistency guarantees that a write is only visible after it is committed durably by the majority quorum of replicas. A write is either synchronously committed durably by both the primary and the quorum of secondaries, or it is aborted. A read is always acknowledged by the majority read quorum, a client can never see an uncommitted or partial write and is always guaranteed to read the latest acknowledged write.
- DocumentDB accounts that are configured to use strong consistency cannot associate more than one Azure region with their DocumentDB account.

- The cost of a read operation (in terms of [request units](#) consumed) with strong consistency is higher than session and eventual, but the same as bounded staleness.

Bounded staleness:

- Bounded staleness consistency guarantees that the reads may lag behind writes by at most K versions or prefixes of a document or t time-interval.
- Consequently, when choosing bounded staleness, the “staleness” can be configured in two ways:
 - Number of versions K of the document by which the reads lag behind the writes
 - Time interval t
- Bounded staleness offers total global order except within the “staleness window”. Note that the monotonic read guarantees exists within a region both inside and outside the “staleness window”.
- Bounded staleness provides a stronger consistency guarantee than session or eventual consistency. For globally distributed applications, we recommend you use bounded staleness for scenarios where you would like to have strong consistency but also want 99.99% availability and low latency.
- DocumentDB accounts that are configured with bounded staleness consistency can associate any number of Azure regions with their DocumentDB account.
- The cost of a read operation (in terms of RUs consumed) with bounded staleness is higher than session and eventual consistency, but the same as strong consistency.

Session:

- Unlike the global consistency models offered by strong and bounded staleness consistency levels, session consistency is scoped to a client session.
- Session consistency is ideal for all scenarios where a device or user session is involved since it guarantees monotonic reads, monotonic writes, and read your own writes (RYW) guarantees.
- Session consistency provides predictable consistency for a session, and maximum read throughput while offering the lowest latency writes and reads.
- DocumentDB accounts that are configured with session consistency can associate any number of Azure regions with their DocumentDB account.
- The cost of a read operation (in terms of RUs consumed) with session consistency level is less than strong and bounded staleness, but more than eventual consistency

Eventual:

- Eventual consistency guarantees that in absence of any further writes, the replicas within the group will eventually converge.
- Eventual consistency is the weakest form of consistency where a client may get the values that are older than the ones it had seen before.
- Eventual consistency provides the weakest read consistency but offers the lowest latency for both reads and writes.
- DocumentDB accounts that are configured with eventual consistency can associate any number of Azure regions with their DocumentDB account.
- The cost of a read operation (in terms of RUs consumed) with the eventual consistency level is the lowest of all the DocumentDB consistency levels.

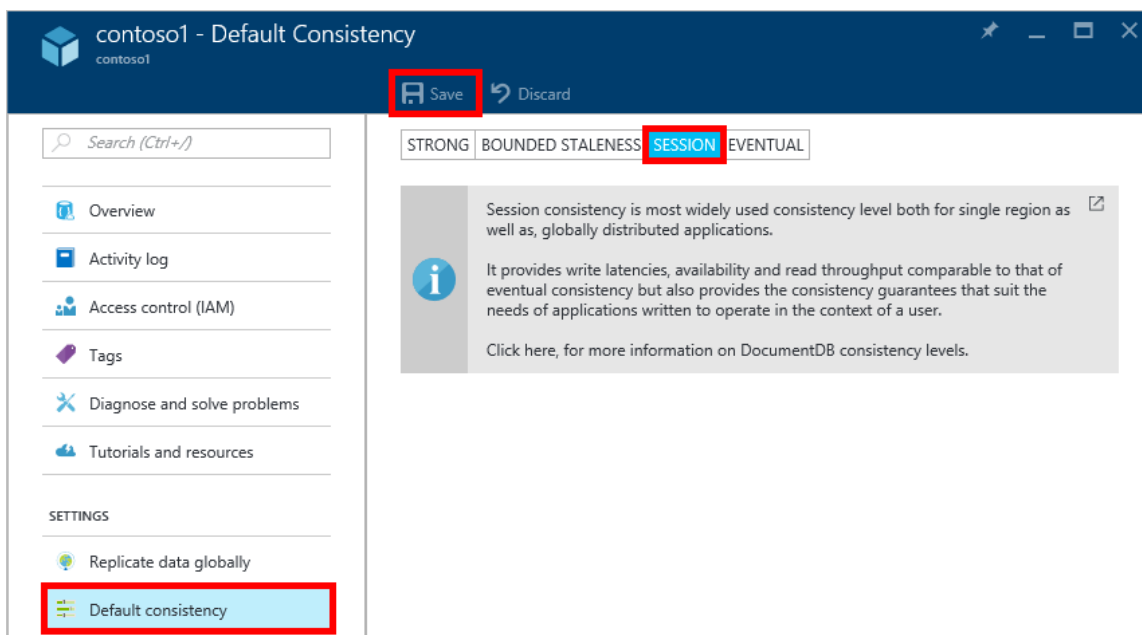
Consistency guarantees

The following table captures various consistency guarantees corresponding to the four consistency levels.

GUARANTEE	STRONG	BOUNDED STALENESS	SESSION	EVENTUAL
Total global order	Yes	Yes, outside of the "staleness window"	No, partial "session" order	No
Consistent prefix guarantee	Yes	Yes	Yes	Yes
Monotonic reads	Yes	Yes, across regions outside of the staleness window and within a region all the time.	Yes, for the given session	No
Monotonic writes	Yes	Yes	Yes	Yes
Read your writes	Yes	Yes	Yes (in the write region)	No

Configuring the default consistency level

1. In the [Azure portal](#), in the Jumpbar, click **DocumentDB (NoSQL)**.
2. In the **DocumentDB (NoSQL)** blade, select the database account to modify.
3. In the account blade, click **Default consistency**.
4. In the **Default Consistency** blade, select the new consistency level and click **Save**.



NOTE

Configuring the default consistency level is not supported within the [Azure DocumentDB Emulator](#).

Consistency levels for queries

By default, for user defined resources, the consistency level for queries is the same as the consistency level for reads. By default, the index is updated synchronously on each insert, replace, or delete of a document to the collection. This enables the queries to honor the same consistency level as that of the document reads. While

DocumentDB is write optimized and supports sustained volumes of document writes, synchronous index maintenance and serving consistent queries, you can configure certain collections to update their index lazily. Lazy indexing further boosts the write performance and is ideal for bulk ingestion scenarios when a workload is primarily read-heavy.

INDEXING MODE	READS	QUERIES
Consistent (default)	Select from strong, bounded staleness, session, or eventual	Select from strong, bounded staleness, session, or eventual
Lazy	Select from strong, bounded staleness, session, or eventual	Eventual
None	Select from strong, bounded staleness, session, or eventual	Not applicable

As with read requests, you can lower the consistency level of a specific query request by specifying the [x-ms-consistency-level](#) request header.

Next steps

If you'd like to do more reading about consistency levels and tradeoffs, we recommend the following resources:

- Doug Terry. Replicated Data Consistency explained through baseball (video).
<https://www.youtube.com/watch?v=glulh8zd26l>
- Doug Terry. Replicated Data Consistency explained through baseball.
<http://research.microsoft.com/pubs/157411/ConsistencyAndBaseballReport.pdf>
- Doug Terry. Session Guarantees for Weakly Consistent Replicated Data.
<http://dl.acm.org/citation.cfm?id=383631>
- Daniel Abadi. Consistency Tradeoffs in Modern Distributed Database Systems Design: CAP is only part of the story".
<http://computer.org/csdl/mags/co/2012/02/mco2012020037-abs.html>
- Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, Ion Stoica. Probabilistic Bounded Staleness (PBS) for Practical Partial Quorums.
http://vldb.org/pvldb/vol5/p776_peterbailis_vldb2012.pdf
- Werner Vogels. Eventual Consistent - Revisited.
http://allthingsdistributed.com/2008/12/eventually_consistent.html

NoSQL vs SQL

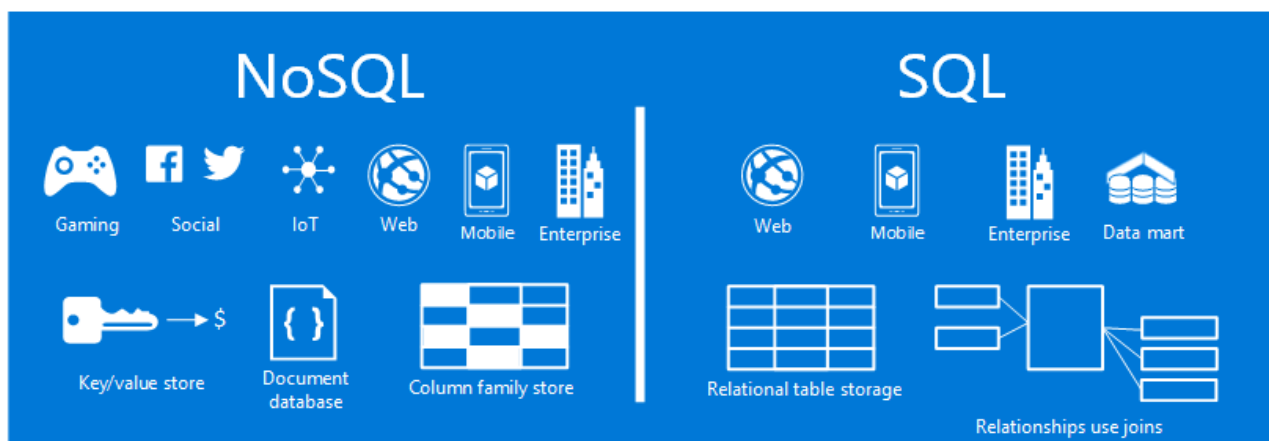
11/15/2016 • 5 min to read • [Edit on GitHub](#)

Contributors

mimig • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [Carl Rabeler](#)

SQL Server and relational databases (RDBMS) have been the go-to databases for over 20 years. However, the increased need to process higher volumes, velocities, and varieties of data at a rapid rate has altered the nature of data storage needs for application developers. In order to enable this scenario, NoSQL databases that enable storing unstructured and heterogeneous data at scale have gained in popularity. For most developers, relational databases are the default or go-to option because a table structure is easy to understand and is familiar, but there are many reasons to explore beyond relational databases.

NoSQL is a category of databases distinctly different from SQL databases. NoSQL is often used to refer to data management systems that are "Not SQL" or an approach to data management that includes "Not only SQL". There are a number of technologies in the NoSQL category, including document databases, key value stores, column family stores, and graph databases, which are popular with gaming, social, and IoT apps.

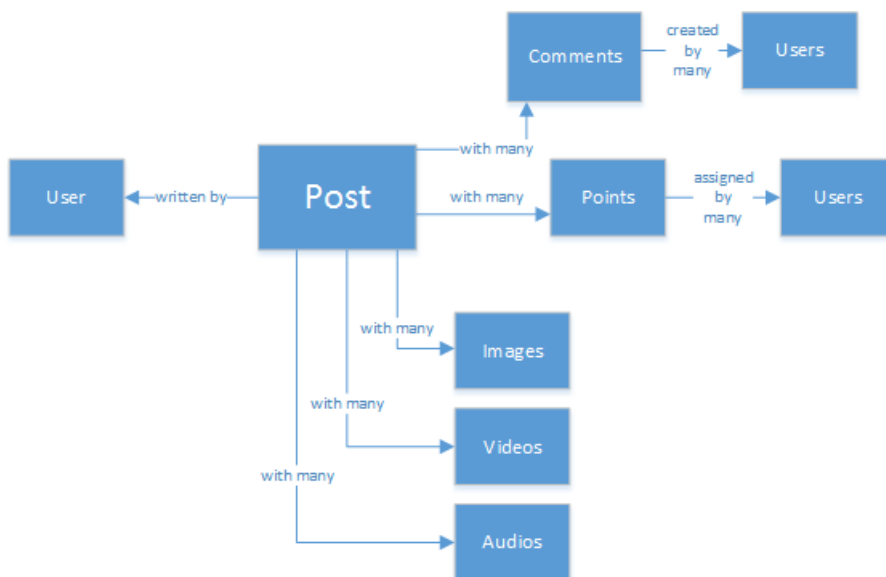


The goal of this article is to help you learn about the differences between NoSQL and SQL, and provide you with an introduction to the NoSQL and SQL offerings from Microsoft.

When to use NoSQL?

Let's imagine you're building a new social engagement site. Users can create posts and add pictures, videos and music to them. Other users can comment on the posts and give points (likes) to rate the posts. The landing page will have a feed of posts that users can share and interact with.

So how do you store this data? If you're familiar with SQL, you might start drawing something like this:



So far, so good, but now think about the structure of a single post and how to display it. If you want to show the post and the associated images, audio, video, comments, points, and user info on a website or application, you'd have to perform a query with eight table joins just to retrieve the content. Now imagine a stream of posts that dynamically load and appear on the screen and you can easily predict that it's going to require thousands of queries and many joins to complete the task.

Now you could use a relational solution like SQL Server to store the data and query it using joins, as SQL supports dynamic data [formatted as JSON](#) - but there's another option, a NoSQL option that simplifies the approach for this specific scenario. By using a single document like the following and storing it in DocumentDB, an Azure NoSQL document database service, you can increase performance and retrieve the whole post with one query and no joins. It's a simpler, more straightforward, and more performant result.

```

{
  "id": "ew12-res2-234e-544f",
  "title": "post title",
  "date": "2016-01-01",
  "body": "this is an awesome post stored on NoSQL",
  "createdBy": "User",
  "images": ["http://myfirstimage.png", "http://mysecondimage.png"],
  "videos": [
    { "url": "http://myfirstvideo.mp4", "title": "The first video" },
    { "url": "http://mysecondvideo.mp4", "title": "The second video" }
  ],
  "audios": [
    { "url": "http://myfirstaudio.mp3", "title": "The first audio" },
    { "url": "http://mysecondaudio.mp3", "title": "The second audio" }
  ]
}

```

In addition, this data can be partitioned by post id allowing the data to scale out naturally and take advantage of NoSQL scale characteristics. Also NoSQL systems allow developers to loosen consistency and offer highly available apps with low-latency. Finally, this solution does not require developers to define, manage and maintain schema in the data tier allowing for rapid iteration.

You can then build on this solution using other Azure services:

- [Azure Search](#) can be used via the web app to enable users to search for posts.
- [Azure App Services](#) can be used to host applications and background processes.
- [Azure Blob Storage](#) can be used to store full user profiles including images.
- [Azure SQL Database](#) can be used to store massive amounts of data such as login information, and data for usage analytics.

- [Azure Machine Learning](#) can be used to build knowledge and intelligence that can provide feedback to the process and help deliver the right content to the right users.

This social engagement site is just one scenario in which a NoSQL database is the right data model for the job. If you're interested in reading more about this scenario and how to model your data for DocumentDB in social media applications, see [Going social with DocumentDB](#).

NoSQL vs SQL comparison

The following table compares the main differences between NoSQL and SQL.

	NoSQL	SQL
Model	Non-relational	Relational
	Stores data in JSON documents, key/value pairs, wide column stores, or graphs	Stores data in a table
Data	Offers flexibility as not every record needs to store the same properties	Great for solutions where every record has the same properties
	New properties can be added on the fly	Adding a new property may require altering schemas or backfilling data
	Relationships are often captured by denormalizing data and presenting all data for an object in a single record	Relationships are often captured in normalized model using joins to resolve references across tables
	Good for semi-structured, complex, or nested data	Good for structured data
Schema	Dynamic or flexible schemas	Strict schema
	Database is schema-agnostic and the schema is dictated by the application. This allows for agility and highly iterative development	Schema must be maintained and kept in sync between application and database
Transactions	ACID transaction support varies per solution	Supports ACID transactions
Consistency & Availability	Eventual to strong consistency supported, depending on solution	Strong consistency enforced
	Consistency, availability, and performance can be traded to meet the needs of the application (CAP theorem)	Consistency is prioritized over availability and performance
Performance	Performance can be maximized by reducing consistency, if needed	Insert and update performance is dependent upon how fast a write is committed, as strong consistency is enforced. Performance can be maximized by using scaling up available resources and using in-memory structures.
	All information about an entity is typically in a single record, so an update can happen in one operation	Information about an entity may be spread across many tables or rows, requiring many joins to complete an update or a query
Scale	Scaling is typically achieved horizontally with data partitioned to span servers	Scaling is typically achieved vertically with more server resources





If a NoSQL database best suits your requirements, continue to the next section to learn more about the NoSQL services available from Azure. Otherwise, if a SQL database best suits your needs, skip to [What are the Microsoft SQL offerings?](#)

What are the Microsoft Azure NoSQL offerings?

Azure has four fully-managed NoSQL services:

- [Azure DocumentDB](#)
- [Azure Table Storage](#)
- [Azure HBase as a part of HDInsight](#)
- [Azure Redis Cache](#)

The following comparison chart maps out the key differentiators for each service. Which one most accurately describes the needs of your application?

	 Azure DocumentDB	 Azure Table Storage	 Azure HBase as part of HDInsight	 Azure Redis Cache
Technology	Document store	Key/value store	Column family store	Redis
Use case	For massive scale applications that need rich queries over a flexible data model, predictable performance, limitless throughput, and/or global distribution to provide low-latency access to any number of regions over a single dataset.	For Internet scale applications requiring large data sets, cost effective storage, and fast time to solution.	For sparsely populated tables that are too big for a relational database. Columns are grouped together into column families.	For a dedicated Redis cache and message broker. Based on the popular open-source Redis project.
Data storage	JSON documents in SSD backed collections	Entities in the form of key-value pairs	Tables, rows, columns, cells, and column families	An in-memory store, with different data structures such as strings, sorted sets, and more
Query language	Enhanced subset of SQL syntax	OData subset	HiveQL and SQL-like syntax	Commands for searching keys and items in a set
Transactional boundary	All documents in the same partition	All entities in the same partition	All cells in same row	Atomic operations on Redis data types
Stored procedure support	Yes in JavaScript	No	Yes in Java	No
Indexing	All properties indexed by default, custom indexes can include or exclude certain paths. Full-text search available through Azure Search.	Indexed by Partition Key and Row Key by default. Secondary indexes can be achieved programmatically. Full-text search available through Azure Search.	RowKey, lexicographically sorted on the primary row key. No secondary indexes.	Use various structures to suit your needs.
Consistency options	Data replicates to any number of regions of your choice with well-defined tunable consistency semantics (at the click of a button): Strong, bounded staleness, session, and eventual	Strong consistency	Strict consistency	Eventual
Performance	<10ms single document reads for 99% of requests <15ms single document writes for 99% of requests 10,000 operations per second per partition	2000 entities/second per partition	Milliseconds for writes and reads, highly tunable	Low latency and high throughput

If one or more of these services might meet the needs of your application, learn more with the following resources:

- [DocumentDB learning path](#) and [DocumentDB use cases](#)
- [Get started with Azure table storage](#)
- [What is HBase in HDInsight](#)
- [Redis Cache learning path](#)

Then go to [Next steps](#) for free trial information.

What are the Microsoft SQL offerings?

Microsoft has five SQL offerings:

- [Azure SQL Database](#)
- [SQL Server on Azure Virtual Machines](#)
- [SQL Server](#)
- [Azure SQL Data Warehouse](#)
- [Analytics Platform System \(on-premises appliance\)](#)

If you're interested in SQL Server on a Virtual Machine or SQL Database, then read [Choose a cloud SQL Server option: Azure SQL \(PaaS\) Database or SQL Server on Azure VMs \(IaaS\)](#) to learn more about the differences between the two.

If SQL sounds like the best option, then go to [SQL Server](#) to learn more about what our Microsoft SQL products and services have to offer.

Then go to [Next steps](#) for free trial and evaluation links.

Next steps

We invite you to learn more about our SQL and NoSQL products by trying them out for free.

- For all Azure services, you can sign up for a [free one-month trial](#) and receive \$200 to spend on any of the Azure services.
 - [Azure DocumentDB](#)
 - [Azure HBase as a part of HDInsight](#)
 - [Azure Redis Cache](#)
 - [Azure SQL Data Warehouse](#)
 - [Azure SQL Database](#)
 - [Azure Table Storage](#)
- You can spin up an [evaluation version of SQL Server 2016 on a virtual machine](#) or download an [evaluation version of SQL Server](#).
 - [SQL Server](#)
 - [SQL Server on Azure Virtual Machines](#)

Import data to DocumentDB with the Database Migration tool

11/22/2016 • 21 min to read • [Edit on GitHub](#)

Contributors

[Andrew Hoh](#) • [mimig](#) • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [Ross McAllister](#) • [Stephen Baron](#)

This article shows you how to use the official open source DocumentDB data migration tool to import data to [Microsoft Azure DocumentDB](#) from various sources, including JSON files, CSV files, SQL, MongoDB, Azure Table storage, Amazon DynamoDB and DocumentDB collections.

After reading this article, you'll be able to answer the following questions:

- How can I import JSON file, CSV file, SQL Server data, or MongoDB data to DocumentDB?
- How can I import data from Azure Table storage, Amazon DynamoDB, and HBase to DocumentDB?
- How can I migrate data between DocumentDB collections?

Prerequisites

Before following the instructions in this article, ensure that you have the following installed:

- [Microsoft .NET Framework 4.51](#) or higher.

Overview of the DocumentDB Data Migration Tool

The DocumentDB Data Migration tool is an open source solution that imports data to DocumentDB from a variety of sources, including:

- JSON files
- MongoDB
- SQL Server
- CSV files
- Azure Table storage
- Amazon DynamoDB
- HBase
- DocumentDB collections

While the import tool includes a graphical user interface (dtui.exe), it can also be driven from the command line (dt.exe). In fact, there is an option to output the associated command after setting up an import through the UI. Tabular source data (e.g. SQL Server or CSV files) can be transformed such that hierarchical relationships (subdocuments) can be created during import. Keep reading to learn more about source options, sample command lines to import from each source, target options, and viewing import results.

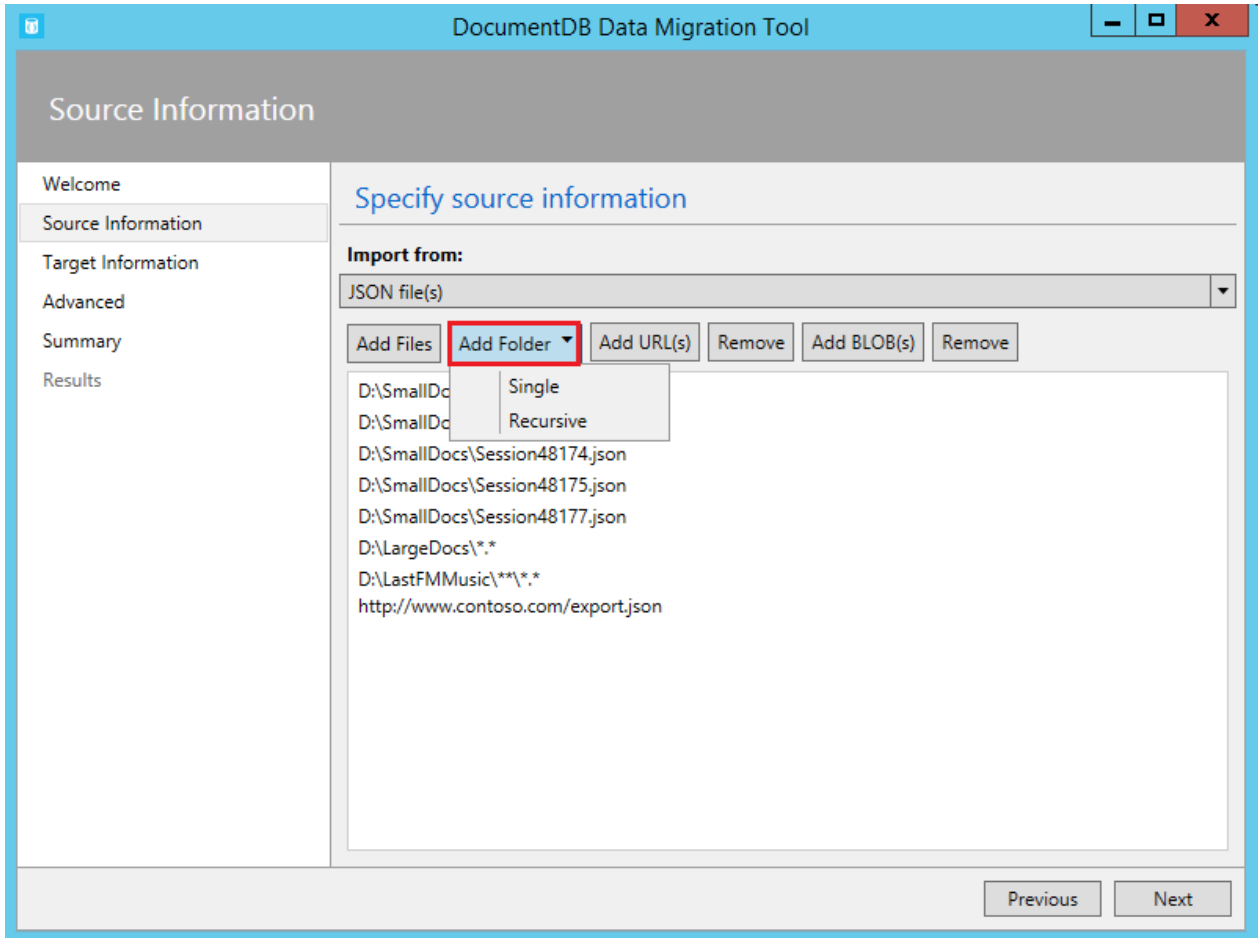
Installing the DocumentDB Data Migration tool

The migration tool source code is available on GitHub in [this repository](#) and a compiled version is available from [Microsoft Download Center](#). You may either compile the solution or simply download and extract the compiled version to a directory of your choice. Then run either:

- **Dtui.exe**: Graphical interface version of the tool
- **Dt.exe**: Command-line version of the tool

Import JSON files

The JSON file source importer option allows you to import one or more single document JSON files or JSON files that each contain an array of JSON documents. When adding folders that contain JSON files to import, you have the option of recursively searching for files in subfolders.



Here are some command line samples to import JSON files:

```
#Import a single JSON file
dt.exe /s:JsonFile /s.Files:.\Sessions.json /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<DocumentDB
Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;" /t.Collection:Sessions
/t.CollectionThroughput:2500

#Import a directory of JSON files
dt.exe /s:JsonFile /s.Files:C:\TESessions\*.json /t:DocumentDBBulk /t.ConnectionString:" AccountEndpoint=
<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;" /t.Collection:Sessions
/t.CollectionThroughput:2500

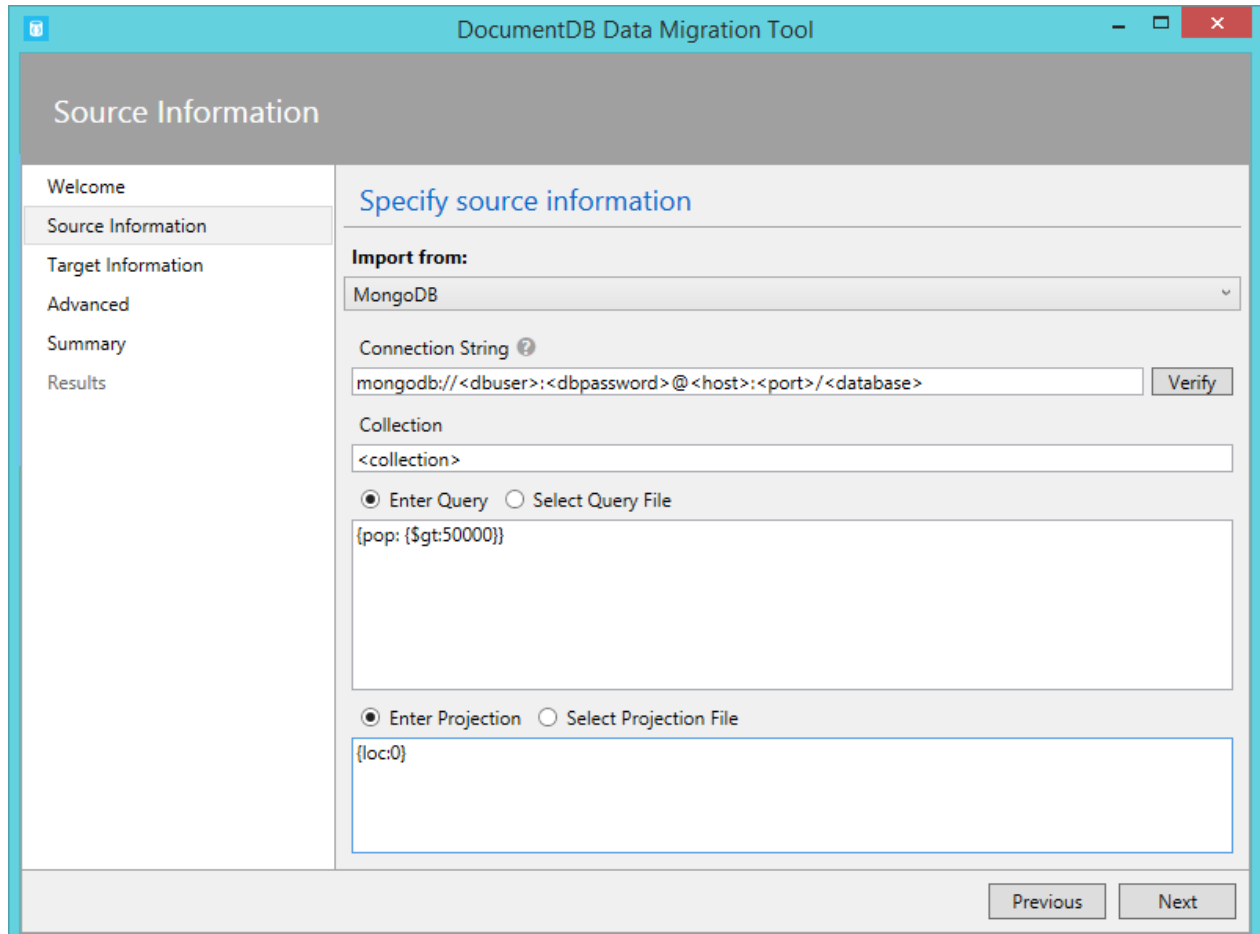
#Import a directory (including sub-directories) of JSON files
dt.exe /s:JsonFile /s.Files:C:\LastFMMusic\**\*.json /t:DocumentDBBulk /t.ConnectionString:" AccountEndpoint=
<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;" /t.Collection:Music
/t.CollectionThroughput:2500

#Import a directory (single), directory (recursive), and individual JSON files
dt.exe /s:JsonFile
/s.Files:C:\Tweets\*.*;C:\LargeDocs\**\*.*;C:\TESessions\Session48172.json;C:\TESessions\Session48173.json;C:\T
ESessions\Session48174.json;C:\TESessions\Session48175.json;C:\TESessions\Session48177.json /t:DocumentDBBulk
/t.ConnectionString:"AccountEndpoint=<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB
Database>;" /t.Collection:subs /t.CollectionThroughput:2500

#Import a single JSON file and partition the data across 4 collections
dt.exe /s:JsonFile /s.Files:D:\\CompanyData\\Companies.json /t:DocumentDBBulk
/t.ConnectionString:"AccountEndpoint=<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB
Database>;" /t.Collection:comp[1-4] /t.PartitionKey.name /t.CollectionThroughput:2500
```

Import from MongoDB

The MongoDB source importer option allows you to import from an individual MongoDB collection and optionally filter documents using a query and/or modify the document structure by using a projection.



The screenshot shows the 'DocumentDB Data Migration Tool' window. On the left is a sidebar with navigation links: 'Welcome', 'Source Information' (selected), 'Target Information', 'Advanced', 'Summary', and 'Results'. The main area is titled 'Specify source information' and contains the following fields and options:

- Import from:** A dropdown menu set to 'MongoDB'.
- Connection String** (with a help icon): A text box containing the template `mongodb://<dbuser>:<dbpassword>@<host>:<port>/<database>` and a 'Verify' button.
- Collection:** A text box containing the placeholder `<collection>`.
- Query Selection:** Two radio buttons: 'Enter Query' (selected) and 'Select Query File'.
- Query:** A text box containing the JSON query `{pop: {$gt:50000}}`.
- Projection Selection:** Two radio buttons: 'Enter Projection' (selected) and 'Select Projection File'.
- Projection:** A text box containing the JSON projection `{loc:0}`.

At the bottom right of the window are 'Previous' and 'Next' buttons.

The connection string is in the standard MongoDB format:

```
mongodb://<dbuser>:<dbpassword>@<host>:<port>/<database>
```

NOTE

Use the Verify command to ensure that the MongoDB instance specified in the connection string field can be accessed.

Enter the name of the collection from which data will be imported. You may optionally specify or provide a file for a query (e.g. {pop: {\$gt:5000}}) and/or projection (e.g. {loc:0}) to both filter and shape the data to be imported.

Here are some command line samples to import from MongoDB:

```
#Import all documents from a MongoDB collection
dt.exe /s:MongoDB /s.ConnectionString:mongodb://<dbuser>:<dbpassword>@<host>:<port>/<database>
/s.Collection:zips /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<DocumentDB Endpoint>;AccountKey=
<DocumentDB Key>;Database=<DocumentDB Database>;" /t.Collection:BulkZips /t.IdField:_id
/t.CollectionThroughput:2500

#Import documents from a MongoDB collection which match the query and exclude the loc field
dt.exe /s:MongoDB /s.ConnectionString:mongodb://<dbuser>:<dbpassword>@<host>:<port>/<database>
/s.Collection:zips /s.Query:{pop:{$gt:50000}} /s.Projection:{loc:0} /t:DocumentDBBulk
/t.ConnectionString:"AccountEndpoint=<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB
Database>;" /t.Collection:BulkZipsTransform /t.IdField:_id/t.CollectionThroughput:2500
```

Import MongoDB export files

The MongoDB export JSON file source importer option allows you to import one or more JSON files produced from the mongoexport utility.

The screenshot shows the 'DocumentDB Data Migration Tool' window. On the left is a sidebar with navigation links: 'Welcome', 'Source Information' (selected), 'Target Information', 'Advanced', 'Summary', and 'Results'. The main area is titled 'Specify source information' and contains an 'Import from:' dropdown menu set to 'MongoDB export (mongoexport) JSON file(s)'. Below this are buttons for 'Add Files', 'Add Folder', 'Add URL(s)', 'Add BLOB(s)', and 'Remove'. A list box below these buttons contains two entries: 'D:\Installation_Files\mongoemployees.json' and 'http://www.contoso.com/exportfiles/mongoexport.json'. At the bottom right of the window are 'Previous' and 'Next' buttons.

When adding folders that contain MongoDB export JSON files for import, you have the option of recursively searching for files in subfolders.

Here is a command line sample to import from MongoDB export JSON files:

```
dt.exe /s:MongoDBExport /s.Files:D:\mongoemployees.json /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=
<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;" /t.Collection:employees
/t.IdField:_id /t.Dates:Epoch /t.CollectionThroughput:2500
```

Import from SQL Server

The SQL source importer option allows you to import from an individual SQL Server database and optionally filter the records to be imported using a query. In addition, you can modify the document structure by specifying a nesting separator (more on that in a moment).

The screenshot shows the 'DocumentDB Data Migration Tool' window. On the left is a sidebar with navigation links: 'Welcome', 'Source Information' (selected), 'Target Information', 'Advanced', 'Summary', and 'Results'. The main area is titled 'Specify source information'. Under 'Import from:', a dropdown menu is set to 'SQL'. Below this is a 'Connection String:' label and a text box containing the format: 'Data Source= <server>;Initial Catalog= <database>;User Id= <user>;Password= <password>;'. To the right of the text box is a 'Verify' button. Below the connection string section are two radio buttons: 'Enter Query' (selected) and 'Select Query File'. Under 'Enter Query' is a large text area containing a SQL query: 'select CAST(BusinessEntityID AS varchar) as Id, Name, AddressType as [Address.AddressType], AddressLine1 as [Address.AddressLine1], City as [Address.Location.City], StateProvinceName as [Address.Location.StateProvinceName], PostalCode as [Address.PostalCode], CountryRegionName as [Address.CountryRegionName] from Sales.vStoreWithAddresses WHERE AddressType= 'Main Office''. Below the query text area is a 'Nesting Separator:' label and a text box containing a period '.'. At the bottom right of the window are 'Previous' and 'Next' buttons.

The format of the connection string is the standard SQL connection string format.

NOTE

Use the Verify command to ensure that the SQL Server instance specified in the connection string field can be accessed.

The nesting separator property is used to create hierarchical relationships (sub-documents) during import. Consider the following SQL query:

```
select CAST(BusinessEntityID AS varchar) as Id, Name, AddressType as [Address.AddressType], AddressLine1 as
[Address.AddressLine1], City as [Address.Location.City], StateProvinceName as
[Address.Location.StateProvinceName], PostalCode as [Address.PostalCode], CountryRegionName as
[Address.CountryRegionName] from Sales.vStoreWithAddresses WHERE AddressType= 'Main Office'
```

Which returns the following (partial) results:

	Id	Name	Address.AddressType	Address.AddressLine1	Address.Location.City	Address.Location.StateProvinceName	Address.PostalCode	Address.CountryRegionName
1	956	Finer Sales and Service	Main Office	#500-75 O'Connor Street	Ottawa	Ontario	K4B 1S2	Canada
2	780	Finer Riding Supplies	Main Office	#9900 2700 Production Way	Burnaby	British Columbia	V5A 4X1	Canada
3	1012	Stylish Department Stores	Main Office	1 Corporate Center Drive	Miami	Florida	33127	United States
4	482	Favourite Toy Distributor	Main Office	1, place de la République	Paris	Seine (Paris)	75017	France
5	1338	Sports Sales and Rental	Main Office	100 Fifth Drive	Millington	Tennessee	38054	United States
6	1424	Closeout Boutique	Main Office	1050 Oak Street	Seattle	Washington	98104	United States
7	1274	Self-Contained Cycle Parts Company	Main Office	12, rue des Grands Champs	Verrières Le Buisson	Essonne	91370	France
8	1958	Ultimate Bicycle Company	Main Office	12, rue Lafayette	Morangis	Essonne	91420	France
9	1110	Local Sales and Rental	Main Office	1200 First Ave.	Joliet	Illinois	60433	United States
10	1262	Roadway Bicycle Supply	Main Office	121, rue de Varenne	Courbevoie	Hauts de Seine	92400	France

Note the aliases such as Address.AddressType and Address.Location.StateProvinceName. By specifying a nesting separator of '.', the import tool creates Address and Address.Location subdocuments during the import. Here is an example of a resulting document in DocumentDB:

```
{ "id": "956", "Name": "Finer Sales and Service", "Address": { "AddressType": "Main Office", "AddressLine1": "#500-75 O'Connor Street", "Location": { "City": "Ottawa", "StateProvinceName": "Ontario" }, "PostalCode": "K4B 1S2", "CountryRegionName": "Canada" } }
```

Here are some command line samples to import from SQL Server:

```
#Import records from SQL which match a query
dt.exe /s:SQL /s.ConnectionString:"Data Source=<server>;Initial Catalog=AdventureWorks;User
Id=advworks;Password=<password>;" /s.Query:"select CAST(BusinessEntityID AS varchar) as Id, * from
Sales.vStoreWithAddresses WHERE AddressType='Main Office'" /t:DocumentDBBulk /t.ConnectionString:"
AccountEndpoint=<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;"
/t.Collection:Stores /t.IdField:Id /t.CollectionThroughput:2500

#Import records from sql which match a query and create hierarchical relationships
dt.exe /s:SQL /s.ConnectionString:"Data Source=<server>;Initial Catalog=AdventureWorks;User
Id=advworks;Password=<password>;" /s.Query:"select CAST(BusinessEntityID AS varchar) as Id, Name, AddressType
as [Address.AddressType], AddressLine1 as [Address.AddressLine1], City as [Address.Location.City],
StateProvinceName as [Address.Location.StateProvinceName], PostalCode as [Address.PostalCode],
CountryRegionName as [Address.CountryRegionName] from Sales.vStoreWithAddresses WHERE AddressType='Main
Office'" /s.NestingSeparator:./ /t:DocumentDBBulk /t.ConnectionString:" AccountEndpoint=<DocumentDB
Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;" /t.Collection:StoresSub /t.IdField:Id
/t.CollectionThroughput:2500
```

Import CSV files - Convert CSV to JSON

The CSV file source importer option enables you to import one or more CSV files. When adding folders that contain CSV files for import, you have the option of recursively searching for files in subfolders.

Similar to the SQL source, the nesting separator property may be used to create hierarchical relationships (sub-documents) during import. Consider the following CSV header row and data rows:

```
DomainInfo.Domain_Name,DomainInfo.Domain_Name_Address,Federal Agency,RedirectInfo.Redirecting,RedirectInfo.Redirect_Destination
ACUS.GOV,http://www.ACUS.GOV,Administrative Conference of the United States,0,
ACHP.GOV,http://www.ACHP.GOV,Advisory Council on Historic Preservation,0,
PRESERVEAMERICA.GOV,http://www.PRESERVEAMERICA.GOV,Advisory Council on Historic Preservation,0,
ADF.GOV,http://www.ADF.GOV,African Development Foundation,0,
```

Note the aliases such as DomainInfo.Domain_Name and RedirectInfo.Redirecting. By specifying a nesting separator of '.', the import tool will create DomainInfo and RedirectInfo subdocuments during the import. Here is an example of a resulting document in DocumentDB:

```
{ "DomainInfo": { "Domain_Name": "ACUS.GOV", "Domain_Name_Address": "http://www.ACUS.GOV" }, "Federal Agency": "Administrative Conference of the United States", "RedirectInfo": { "Redirecting": "0", "Redirect_Destination": "" }, "id": "9cc565c5-ebcd-1c03-ebd3-cc3e2ecd814d" }
```

The import tool will attempt to infer type information for unquoted values in CSV files (quoted values are always treated as strings). Types are identified in the following order: number, datetime, boolean.

There are two other things to note about CSV import:

1. By default, unquoted values are always trimmed for tabs and spaces, while quoted values are preserved as-is. This behavior can be overridden with the Trim quoted values checkbox or the `/s.TrimQuoted` command line option.
2. By default, an unquoted null is treated as a null value. This behavior can be overridden (i.e. treat an unquoted null as a "null" string) with the Treat unquoted NULL as string checkbox or the `/s.NoUnquotedNulls` command line option.

Here is a command line sample for CSV import:

```
dt.exe /s:CsvFile /s.Files:.\Employees.csv /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<DocumentDB
Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;" /t.Collection:Employees
/t.IdField:EntityID /t.CollectionThroughput:2500
```

Import from Azure Table storage

The Azure Table storage source importer option allows you to import from an individual Azure Table storage table and optionally filter the table entities to be imported.

The screenshot shows the 'DocumentDB Data Migration Tool' window. On the left is a sidebar with navigation links: 'Welcome', 'Source Information' (selected), 'Target Information', 'Advanced', 'Summary', and 'Results'. The main area is titled 'Specify source information' and contains the following fields and controls:

- Import from:** A dropdown menu set to 'Azure Table'.
- Connection String:** A text box containing 'DefaultEndpointsProtocol=https;AccountName=<Account Name>;AccountKey=<Account Key>;' with a 'Verify' button to its right.
- Table Name:** A text box containing 'metrics'.
- Include Internal Fields:** A dropdown menu set to 'All'.
- Filter:** A text box containing 'PartitionKey eq 'Partition1' and RowKey gt '00001''.
- Select Columns:** A section with a text box, 'Add', and 'Remove' buttons. Below it, a list box contains 'ObjectCount' and 'ObjectSize'.

At the bottom right of the main area are 'Previous' and 'Next' buttons.

The format of the Azure Table storage connection string is:

```
DefaultEndpointsProtocol=<protocol>;AccountName=<Account Name>;AccountKey=<Account Key>;
```

NOTE

Use the Verify command to ensure that the Azure Table storage instance specified in the connection string field can be accessed.

Enter the name of the Azure table from which data will be imported. You may optionally specify a [filter](#).

The Azure Table storage source importer option has the following additional options:

1. Include Internal Fields
 - a. All - Include all internal fields (PartitionKey, RowKey, and Timestamp)
 - b. None - Exclude all internal fields
 - c. RowKey - Only include the RowKey field
2. Select Columns
 - a. Azure Table storage filters do not support projections. If you want to only import specific Azure Table

entity properties, add them to the Select Columns list. All other entity properties will be ignored.

Here is a command line sample to import from Azure Table storage:

```
dt.exe /s:AzureTable /s.ConnectionString:"DefaultEndpointsProtocol=https;AccountName=<Account Name>;AccountKey=
<Account Key>" /s.Table:metrics /s.InternalFields:All /s.Filter:"PartitionKey eq 'Partition1' and RowKey gt
'00001'" /s.Projection:ObjectCount;ObjectSize /t:DocumentDBBulk /t.ConnectionString:" AccountEndpoint=
<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;" /t.Collection:metrics
/t.CollectionThroughput:2500
```

Import from Amazon DynamoDB

The Amazon DynamoDB source importer option allows you to import from an individual Amazon DynamoDB table and optionally filter the entities to be imported. Several templates are provided so that setting up an import is as easy as possible.

The screenshot shows the 'DocumentDB Data Migration Tool' window. The 'Source Information' tab is selected in the left sidebar. The main area is titled 'Specify source information'. Under 'Import from:', 'DynamoDB' is selected in a dropdown menu. The 'Connection String' field is populated with 'ServiceURL=https://dynamodb.us-east-1.amazonaws.com;AccessKey= <accessKey>;SecretKey=' and a 'Verify' button is next to it. Below this, there are two radio buttons: 'Enter Request' (selected) and 'Select Request File'. A large red rectangular box highlights a context menu that appears over the main area. The menu has a 'Templates' header with a right-pointing arrow. Below it are three items: 'Copy' with 'Ctrl+C', 'Cut' with 'Ctrl+X', and 'Paste' with 'Ctrl+V'. To the right of these items are three more options: 'Scan', 'Filtered Scan', and 'Filtered Query'.

Templates	
Copy	Ctrl+C
Cut	Ctrl+X
Paste	Ctrl+V

Scan
Filtered Scan
Query
Filtered Query

Previous Next

The format of the Amazon DynamoDB connection string is:

```
ServiceURL=<Service Address>;AccessKey=<Access Key>;SecretKey=<Secret Key>;
```

NOTE

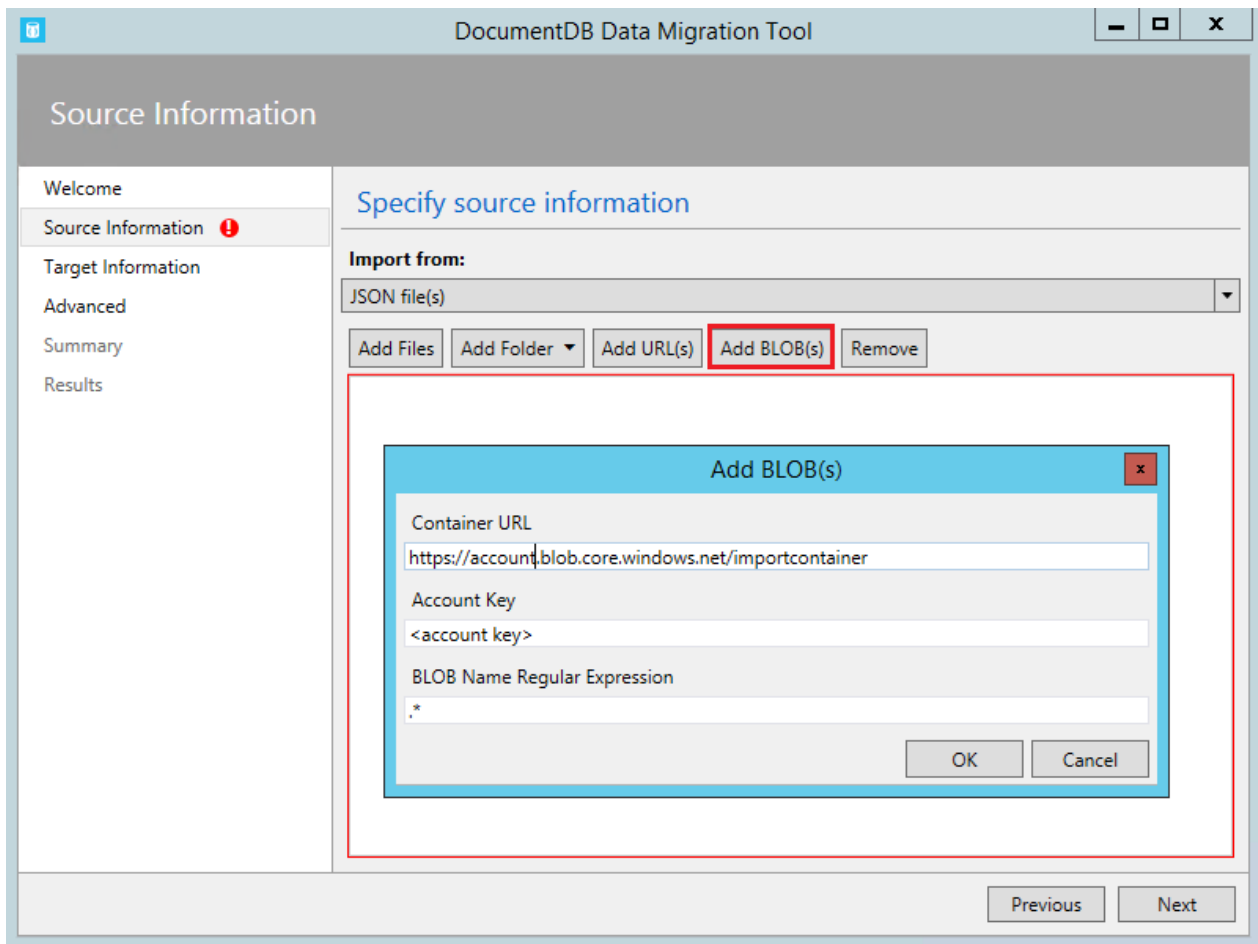
Use the Verify command to ensure that the Amazon DynamoDB instance specified in the connection string field can be accessed.

Here is a command line sample to import from Amazon DynamoDB:

```
dt.exe /s:DynamoDB /s.ConnectionString:ServiceURL=https://dynamodb.us-east-1.amazonaws.com;AccessKey=
<accessKey>;SecretKey=<secretKey> /s.Request:"{  ""TableName"": ""ProductCatalog"" }" /t:DocumentDBBulk
/t.ConnectionString:"AccountEndpoint=<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB
Database>;" /t.Collection:catalogCollection /t.CollectionThroughput:2500
```

Import files from Azure Blob storage

The JSON file, MongoDB export file, and CSV file source importer options allow you to import one or more files from Azure Blob storage. After specifying a Blob container URL and Account Key, simply provide a regular expression to select the file(s) to import.



Here is command line sample to import JSON files from Azure Blob storage:

```
dt.exe /s:JsonFile /s.Files:"blobs://<account key>@account.blob.core.windows.net:443/importcontainer/.*"
/t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<DocumentDB Endpoint>;AccountKey=<DocumentDB
Key>;Database=<DocumentDB Database>;" /t.Collection:doctest
```

Import from DocumentDB

The DocumentDB source importer option allows you to import data from one or more DocumentDB collections and optionally filter documents using a query.

The format of the DocumentDB connection string is:

```
AccountEndpoint=<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;
```

The DocumentDB account connection string can be retrieved from the Keys blade of the Azure portal, as described in [How to manage a DocumentDB account](#), however the name of the database needs to be appended to the connection string in the following format:

```
Database=<DocumentDB Database>;
```

NOTE

Use the Verify command to ensure that the DocumentDB instance specified in the connection string field can be accessed.

To import from a single DocumentDB collection, enter the name of the collection from which data will be imported. To import from multiple DocumentDB collections, provide a regular expression to match one or more collection names (e.g. collection01 | collection02 | collection03). You may optionally specify, or provide a file for, a query to both filter and shape the data to be imported.

NOTE

Since the collection field accepts regular expressions, if you are importing from a single collection whose name contains regular expression characters, then those characters must be escaped accordingly.

The DocumentDB source importer option has the following advanced options:

1. Include Internal Fields: Specifies whether or not to include DocumentDB document system properties in the

export (e.g. _rid, _ts).

2. Number of Retries on Failure: Specifies the number of times to retry the connection to DocumentDB in case of transient failures (e.g. network connectivity interruption).
3. Retry Interval: Specifies how long to wait between retrying the connection to DocumentDB in case of transient failures (e.g. network connectivity interruption).
4. Connection Mode: Specifies the connection mode to use with DocumentDB. The available choices are DirectTcp, DirectHttps, and Gateway. The direct connection modes are faster, while the gateway mode is more firewall friendly as it only uses port 443.

DocumentDB Data Migration Tool

Source Information

Welcome

Source Information

Target Information

Advanced

Summary

Results

Specify source information

Import from:

DocumentDB

☒ Enter Query ☐ Select Query File

SELECT * FROM c WHERE c.isActive = true

Advanced Options

☐ Include Internal Fields 1

Number of Retries on Failure 2

10

Retry Interval 3

00:00:01

Connection Mode ? 4

DirectTcp

Previous Next

TIP

The import tool defaults to connection mode DirectTcp. If you experience firewall issues, switch to connection mode Gateway, as it only requires port 443.

Here are some command line samples to import from DocumentDB:

```
#Migrate data from one DocumentDB collection to another DocumentDB collections
dt.exe /s:DocumentDB /s.ConnectionString:"AccountEndpoint=<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;" /s.Collection:TEColl /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;" /t.Collection:TESessions /t.CollectionThroughput:2500

#Migrate data from multiple DocumentDB collections to a single DocumentDB collection
dt.exe /s:DocumentDB /s.ConnectionString:"AccountEndpoint=<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;" /s.Collection:comp1|comp2|comp3|comp4 /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;" /t.Collection:singleCollection /t.CollectionThroughput:2500

#Export a DocumentDB collection to a JSON file
dt.exe /s:DocumentDB /s.ConnectionString:"AccountEndpoint=<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;" /s.Collection:StoresSub /t:JsonFile /t.File:StoresExport.json /t.Overwrite /t.CollectionThroughput:2500
```

TIP

The DocumentDB Data Import Tool also supports import of data from the [DocumentDB Emulator](#). When importing data from a local emulator, set the endpoint to <https://localhost:>.

Import from HBase

The HBase source importer option allows you to import data from an HBase table and optionally filter the data. Several templates are provided so that setting up an import is as easy as possible.

The screenshot shows the 'DocumentDB Data Migration Tool' window. The 'Source Information' tab is selected in the left sidebar. The main area is titled 'Specify source information' and contains the following fields and controls:

- Import from:** A dropdown menu with 'HBase' selected.
- Connection String ?**: A text input field with the placeholder 'ServiceURL= <server-address>;Username= <username>;Password= <password>'. A 'Verify' button is to the right.
- Table Name**: A text input field containing 'Contacts'.
- Filter Options**: Two radio buttons, 'Enter Filter' (selected) and 'Select Filter File'.
- Exclude Row Id**: An unchecked checkbox.
- Batch Size**: A text input field containing '100'.

At the bottom right, there are 'Previous' and 'Next' buttons.

The format of the HBase Stargate connection string is:

```
ServiceURL=<server-address>;Username=<username>;Password=<password>
```

NOTE

Use the Verify command to ensure that the HBase instance specified in the connection string field can be accessed.

Here is a command line sample to import from HBase:

```
dt.exe /s:HBase /s.ConnectionString:ServiceURL=<server-address>;Username=<username>;Password=<password>
/s.Table:Contacts /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<DocumentDB Endpoint>;AccountKey=
<DocumentDB Key>;Database=<DocumentDB Database>;" /t.Collection:hbaseimport
```

Import to DocumentDB (Bulk Import)

The DocumentDB Bulk importer allows you to import from any of the available source options, using a DocumentDB stored procedure for efficiency. The tool supports import to one single-partitioned DocumentDB collection, as well as sharded import whereby data is partitioned across multiple single-partitioned DocumentDB collections. For more information about partitioning data, see [Partitioning and scaling in Azure DocumentDB](#). The tool will create, execute, and then delete the stored procedure from the target collection(s).

The format of the DocumentDB connection string is:

```
AccountEndpoint=<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;
```

The DocumentDB account connection string can be retrieved from the Keys blade of the Azure portal, as described in [How to manage a DocumentDB account](#), however the name of the database needs to be appended to the connection string in the following format:

```
Database=<DocumentDB Database>;
```

NOTE

Use the Verify command to ensure that the DocumentDB instance specified in the connection string field can be accessed.

To import to a single collection, enter the name of the collection to which data will be imported and click the Add button. To import to multiple collections, either enter each collection name individually or use the following syntax to specify multiple collections: *collection_prefix*[start index - end index]. When specifying multiple collections via the aforementioned syntax, keep the following in mind:

1. Only integer range name patterns are supported. For example, specifying collection[0-3] will produce the following collections: collection0, collection1, collection2, collection3.
2. You can use an abbreviated syntax: collection[3] will emit same set of collections mentioned in step 1.
3. More than one substitution can be provided. For example, collection[0-1] [0-9] will generate 20 collection names with leading zeros (collection01, ..02, ..03).

Once the collection name(s) have been specified, choose the desired throughput of the collection(s) (400 RUs to 10,000 RUs). For best import performance, choose a higher throughput. For more information about performance levels, see [Performance levels in DocumentDB](#).

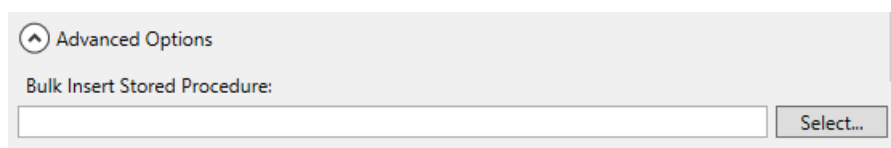
NOTE

The performance throughput setting only applies to collection creation. If the specified collection already exists, its throughput will not be modified.

When importing to multiple collections, the import tool supports hash based sharding. In this scenario, specify the document property you wish to use as the Partition Key (if Partition Key is left blank, documents will be sharded randomly across the target collections).

You may optionally specify which field in the import source should be used as the DocumentDB document id property during the import (note that if documents do not contain this property, then the import tool will generate a GUID as the id property value).

There are a number of advanced options available during import. First, while the tool includes a default bulk import stored procedure (BulkInsert.js), you may choose to specify your own import stored procedure:

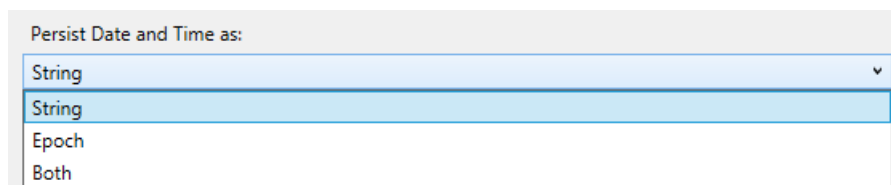


Advanced Options

Bulk Insert Stored Procedure:

Select...

Additionally, when importing date types (e.g. from SQL Server or MongoDB), you can choose between three import options:



Persist Date and Time as:

- String
- String
- Epoch
- Both

- String: Persist as a string value
- Epoch: Persist as an Epoch number value
- Both: Persist both string and Epoch number values. This option will create a subdocument, for example: `"date_joined": { "Value": "2013-10-21T21:17:25.2410000Z", "Epoch": 1382390245 }`

The DocumentDB Bulk importer has the following additional advanced options:

1. Batch Size: The tool defaults to a batch size of 50. If the documents to be imported are large, consider lowering the batch size. Conversely, if the documents to be imported are small, consider raising the batch size.
2. Max Script Size (bytes): The tool defaults to a max script size of 512KB
3. Disable Automatic Id Generation: If every document to be imported contains an id field, then selecting this option can increase performance. Documents missing a unique id field will not be imported.
4. Update Existing Documents: The tool defaults to not replacing existing documents with id conflicts. Selecting this option will allow overwriting existing documents with matching ids. This feature is useful for scheduled data migrations that update existing documents.
5. Number of Retries on Failure: Specifies the number of times to retry the connection to DocumentDB in case of transient failures (e.g. network connectivity interruption).
6. Retry Interval: Specifies how long to wait between retrying the connection to DocumentDB in case of transient failures (e.g. network connectivity interruption).
7. Connection Mode: Specifies the connection mode to use with DocumentDB. The available choices are DirectTcp, DirectHttps, and Gateway. The direct connection modes are faster, while the gateway mode is more firewall friendly as it only uses port 443.

DocumentDB Data Migration Tool

Target Information

Welcome

Source Information

Target Information

Advanced

Summary

Results

Specify target information

Export to:

DocumentDB - Bulk import (legacy collections)

Advanced Options

Bulk Insert Stored Procedure

Select...

Batch Size

50

Max Script Size (bytes)

524278

☐ Disable Automatic Id Generation

☐ Update Existing Documents

Persist Date and Time as

String

☒ Enter Indexing Policy
☐ Select Policy File

Number of Retries on Failure

10

Retry Interval

00:00:01

Connection Mode

DirectTcp

Previous

Next

TIP

The import tool defaults to connection mode DirectTcp. If you experience firewall issues, switch to connection mode Gateway, as it only requires port 443.

Import to DocumentDB (Sequential Record Import)

The DocumentDB sequential record importer allows you to import from any of the available source options on a record by record basis. You might choose this option if you're importing to an existing collection that has reached its quota of stored procedures. The tool supports import to a single (both single-partition and multi-partition) DocumentDB collection, as well as sharded import whereby data is partitioned across multiple single-partition and/or multi-partition DocumentDB collections. For more information about partitioning data, see [Partitioning and scaling in Azure DocumentDB](#).

The format of the DocumentDB connection string is:

```
AccountEndpoint=<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;
```

The DocumentDB account connection string can be retrieved from the Keys blade of the Azure portal, as described in [How to manage a DocumentDB account](#), however the name of the database needs to be appended to the connection string in the following format:

```
Database=<DocumentDB Database>;
```

NOTE

Use the Verify command to ensure that the DocumentDB instance specified in the connection string field can be accessed.

To import to a single collection, enter the name of the collection to which data will be imported and click the Add button. To import to multiple collections, either enter each collection name individually or use the following syntax to specify multiple collections: *collection_prefix*[start index - end index]. When specifying multiple collections via the aforementioned syntax, keep the following in mind:

1. Only integer range name patterns are supported. For example, specifying collection[0-3] will produce the following collections: collection0, collection1, collection2, collection3.
2. You can use an abbreviated syntax: collection[3] will emit same set of collections mentioned in step 1.
3. More than one substitution can be provided. For example, collection[0-1] [0-9] will generate 20 collection names with leading zeros (collection01, ..02, ..03).

Once the collection name(s) have been specified, choose the desired throughput of the collection(s) (400 RUs to 250,000 RUs). For best import performance, choose a higher throughput. For more information about

performance levels, see [Performance levels in DocumentDB](#). Any import to collections with throughput > 10,000 RUs will require a partition key. If you choose to have more than 250,000 RUs, see [Request increased DocumentDB account limits](#).

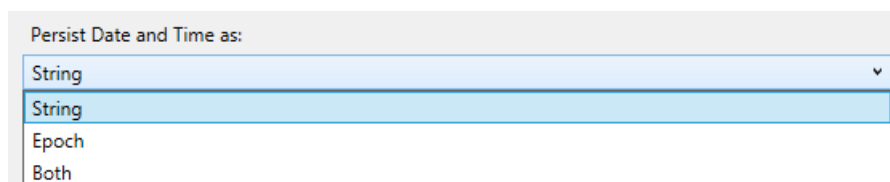
NOTE

The throughput setting only applies to collection creation. If the specified collection already exists, its throughput will not be modified.

When importing to multiple collections, the import tool supports hash based sharding. In this scenario, specify the document property you wish to use as the Partition Key (if Partition Key is left blank, documents will be sharded randomly across the target collections).

You may optionally specify which field in the import source should be used as the DocumentDB document id property during the import (note that if documents do not contain this property, then the import tool will generate a GUID as the id property value).

There are a number of advanced options available during import. First, when importing date types (e.g. from SQL Server or MongoDB), you can choose between three import options:



The image shows a dropdown menu titled "Persist Date and Time as:". The menu is open, displaying four options: "String", "String", "Epoch", and "Both". The first "String" option is highlighted with a blue background and a white downward arrow on the right side.

- String: Persist as a string value
- Epoch: Persist as an Epoch number value
- Both: Persist both string and Epoch number values. This option will create a subdocument, for example: `"date_joined": { "Value": "2013-10-21T21:17:25.2410000Z", "Epoch": 1382390245 }`

The DocumentDB - Sequential record importer has the following additional advanced options:

1. Number of Parallel Requests: The tool defaults to 2 parallel requests. If the documents to be imported are small, consider raising the number of parallel requests. Note that if this number is raised too much, the import may experience throttling.
2. Disable Automatic Id Generation: If every document to be imported contains an id field, then selecting this option can increase performance. Documents missing a unique id field will not be imported.
3. Update Existing Documents: The tool defaults to not replacing existing documents with id conflicts. Selecting this option will allow overwriting existing documents with matching ids. This feature is useful for scheduled data migrations that update existing documents.
4. Number of Retries on Failure: Specifies the number of times to retry the connection to DocumentDB in case of transient failures (e.g. network connectivity interruption).
5. Retry Interval: Specifies how long to wait between retrying the connection to DocumentDB in case of transient failures (e.g. network connectivity interruption).
6. Connection Mode: Specifies the connection mode to use with DocumentDB. The available choices are DirectTcp, DirectHttps, and Gateway. The direct connection modes are faster, while the gateway mode is more firewall friendly as it only uses port 443.

DocumentDB Data Migration Tool

Target Information

Welcome

Source Information

Target Information

Advanced

Summary

Results

Specify target information

Export to:

DocumentDB - Sequential record import (partitioned collection)

Advanced Options

Number of Parallel Requests **1**

2

☐ Disable Automatic Id Generation **2**

☐ Update Existing Documents **3**

Persist Date and Time as

String

☒ Enter Indexing Policy ☐ Select Policy File

Number of Retries on Failure **4**

10

Retry Interval **5**

00:00:01

Connection Mode **6**

DirectTcp

Previous Next

TIP

The import tool defaults to connection mode DirectTcp. If you experience firewall issues, switch to connection mode Gateway, as it only requires port 443.

Specify an indexing policy when creating DocumentDB collections

When you allow the migration tool to create collections during import, you can specify the indexing policy of the collections. In the advanced options section of the DocumentDB Bulk import and DocumentDB Sequential record options, navigate to the Indexing Policy section.

DocumentDB Data Migration Tool

Target Information

Welcome

Source Information

Target Information

Advanced

Summary

Results

Specify target information

Export to:

DocumentDB - Bulk import (legacy collections)

50

Max Script Size (bytes)

524280

☐ Disable Automatic Id Generation ?

Persist Date and Time as

String

☒ Enter Indexing Policy ☐ Select Policy File

Number of Retries on Failure

10

Previous Next

Using the Indexing Policy advanced option, you can select an indexing policy file, manually enter an indexing policy, or select from a set of default templates (by right clicking in the indexing policy textbox).

The policy templates the tool provides are:

- Default. This policy is best when you're performing equality queries against strings and using ORDER BY, range, and equality queries for numbers. This policy has a lower index storage overhead than Range.
- Range. This policy is best you're using ORDER BY, range and equality queries on both numbers and strings. This policy has a higher index storage overhead than Default or Hash.

DocumentDB Data Migration Tool

Target Information

Welcome

Source Information

Target Information

Advanced

Summary

Results

Specify target information

Export to:

DocumentDB - Bulk import (legacy collections)

Batch Size

50

Max Script Size (bytes)

524278

☐ Disable Automatic Id Generation ?

☐ Update Existing Documents ?

Persist Date and Time as

String

☒ Enter Indexing Policy ☐ Select Policy File

Default

Range

CopyCtrl+C

CutCtrl+X

PasteCtrl+V

Number of Retries on Failure

10

Retry Interval

00:00:01

Connection Mode ?

DirectTcp

Previous

Next

NOTE

If you do not specify an indexing policy, then the default policy will be applied. For more information about indexing policies, see [DocumentDB indexing policies](#).

Export to JSON file

The DocumentDB JSON exporter allows you to export any of the available source options to a JSON file that contains an array of JSON documents. The tool will handle the export for you, or you can choose to view the resulting migration command and run the command yourself. The resulting JSON file may be stored locally or in Azure Blob storage.

DocumentDB Data Migration Tool

Target Information

Welcome

Source Information

Target Information

Advanced

Summary

Results

Specify target information

Export to:

JSON file

☒ Local file

☐ BLOB

c:\myExport.json

Select...

☒ Prettify JSON

Previous

Next

DocumentDB Data Migration Tool

Target Information

Welcome

Source Information

Target Information

Advanced

Summary

Results

Specify target information

Export to:

JSON file

☐ Local file

☒ BLOB

Container URL

https://account.blob.core.windows.net/export/

Account Key

<Account Key>

BLOB Name

myexport.json

☐ Overwrite if exists

☒ Prettify JSON

Previous

Next

You may optionally choose to prettify the resulting JSON, which will increase the size of the resulting document while making the contents more human readable.

Standard JSON export

```
[{"id":"Sample","Title":"About Paris","Language":{"Name":"English"},"Author":{"Name":"Don","Location":{"City":"Paris","Country":"France"}}, "Content":"Don's document in DocumentDB is a valid JSON document as defined by the JSON spec.", "PageViews":10000,"Topics":[{"Title":"History of Paris"}, {"Title":"Places to see in Paris"}]}
```

Prettified JSON export

```
[
  {
    "id": "Sample",
    "Title": "About Paris",
    "Language": {
      "Name": "English"
    },
    "Author": {
      "Name": "Don",
      "Location": {
        "City": "Paris",
        "Country": "France"
      }
    },
    "Content": "Don's document in DocumentDB is a valid JSON document as defined by the JSON spec.",
    "PageViews": 10000,
    "Topics": [
      {
        "Title": "History of Paris"
      },
      {
        "Title": "Places to see in Paris"
      }
    ]
  }
]
```

Advanced configuration

In the Advanced configuration screen, specify the location of the log file to which you would like any errors written. The following rules apply to this page:

1. If a file name is not provided, then all errors will be returned on the Results page.
2. If a file name is provided without a directory, then the file will be created (or overwritten) in the current environment directory.
3. If you select an existing file, then the file will be overwritten, there is no append option.

Then, choose whether to log all, critical, or no error messages. Finally, decide how frequently the on screen transfer message will be updated with its progress.

![Screenshot of Advanced configuration screen](./media/documentdb-import-data/AdvancedConfiguration.png)

Confirm import settings and view command line

1. After specifying source information, target information, and advanced configuration, review the migration summary and, optionally, view/copy the resulting migration command (copying the command is useful to automate import operations):

DocumentDB Data Migration Tool

Summary

Welcome

Source Information

Target Information

Advanced

Summary

Results

Confirm import settings

View Command

Source (MongoDB)

Connection String:

Collection:

Query:

Projection:

Sink (DocumentDB - Bulk import)

Connection String:

Collection:

Id Field:

Bulk Insert Stored Procedure:

Batch Size:

Max Script Size (bytes):

Disable Automatic Id Generation:

Persist Date and Time as:

Number of Retries on Failure:

Retry Interval:

Previous

Import

DocumentDB Data Migration Tool

Summary

Welcome

Source Information

Target Information

Advanced

Summary

Results

Confirm import settings

View Command

Source (MongoDB)

Connection String:

Collection:

Bulk Insert Stored Procedure:

Batch Size:

Max Script Size (bytes):

Disable Automatic Id Generation:

Persist Date and Time as:

Number of Retries on Failure:

Retry Interval:

Previous

Import

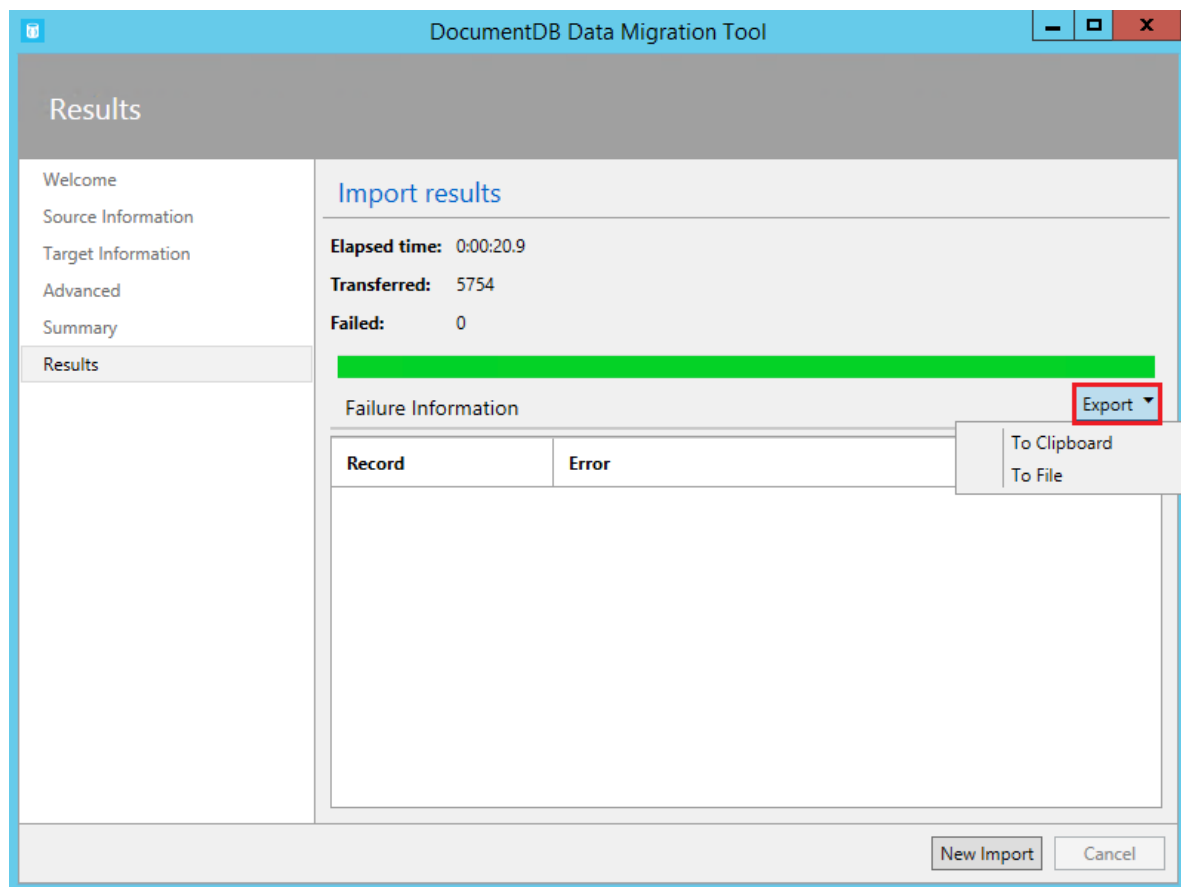
Command Line Preview

```
/s:MongoDB /s.ConnectionString:mongodb://<dbuser>:<dbpassword>@<host>:<port>/<database> /s.Collection:zips /s.Query:{"pop: {$gt:5000}}" /s.Projection:{loc:0} /t.DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;" /t.Collection:zips /t.IdField:_id /t.DisableIdGeneration /t.Dates:Epoch
```

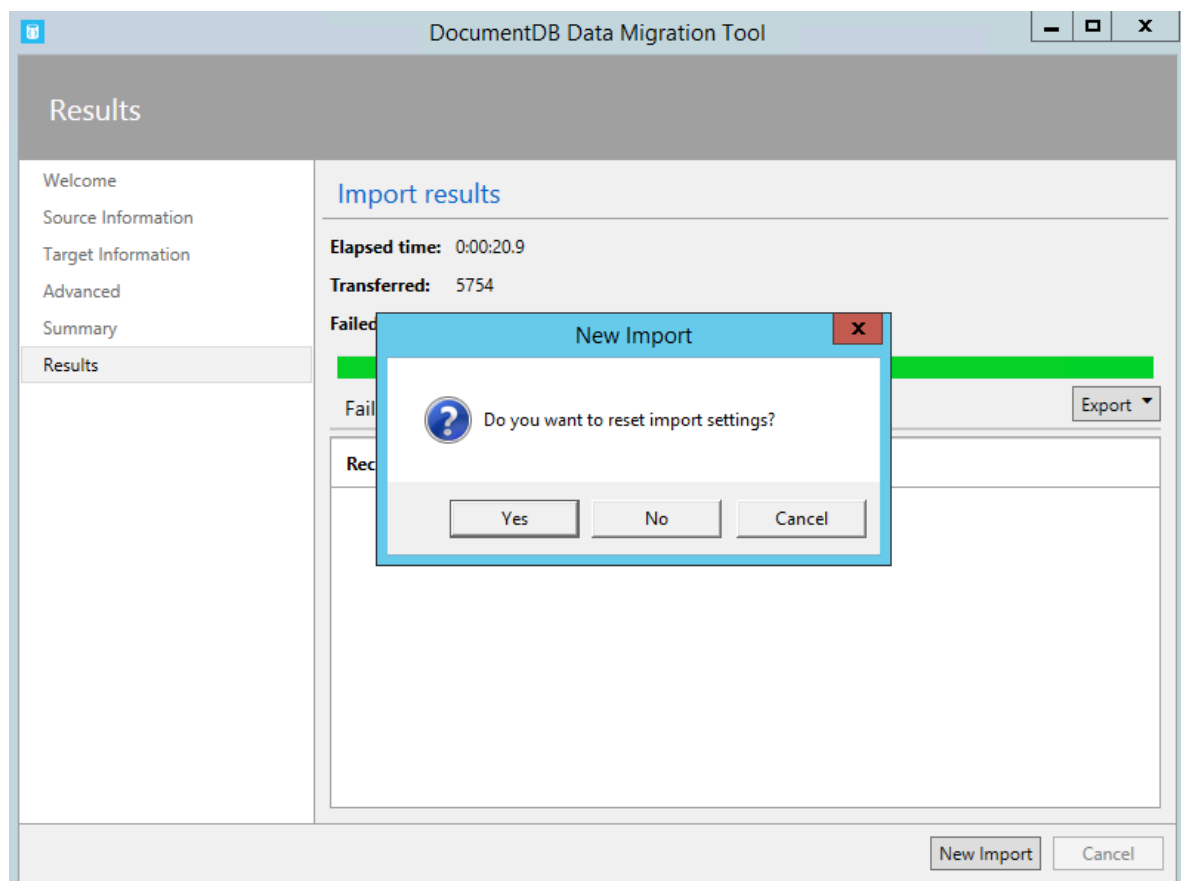
Copy

Close

2. Once you're satisfied with your source and target options, click **Import**. The elapsed time, transferred count, and failure information (if you didn't provide a file name in the Advanced configuration) will update as the import is in process. Once complete, you can export the results (e.g. to deal with any import failures).



3. You may also start a new import, either keeping the existing settings (e.g. connection string information, source and target choice, etc.) or resetting all values.



Next steps

- To learn more about DocumentDB, see the [Learning Path](#).

Modeling data in DocumentDB

11/15/2016 • 14 min to read • [Edit on GitHub](#)

Contributors

Kirat Pandya • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • mimig • James Dunn • James Orr • Rohan
• Stephen Baron • Ryan CrawCour

While schema-free databases, like Azure DocumentDB, make it super easy to embrace changes to your data model you should still spend some time thinking about your data.

How is data going to be stored? How is your application going to retrieve and query data? Is your application read heavy, or write heavy?

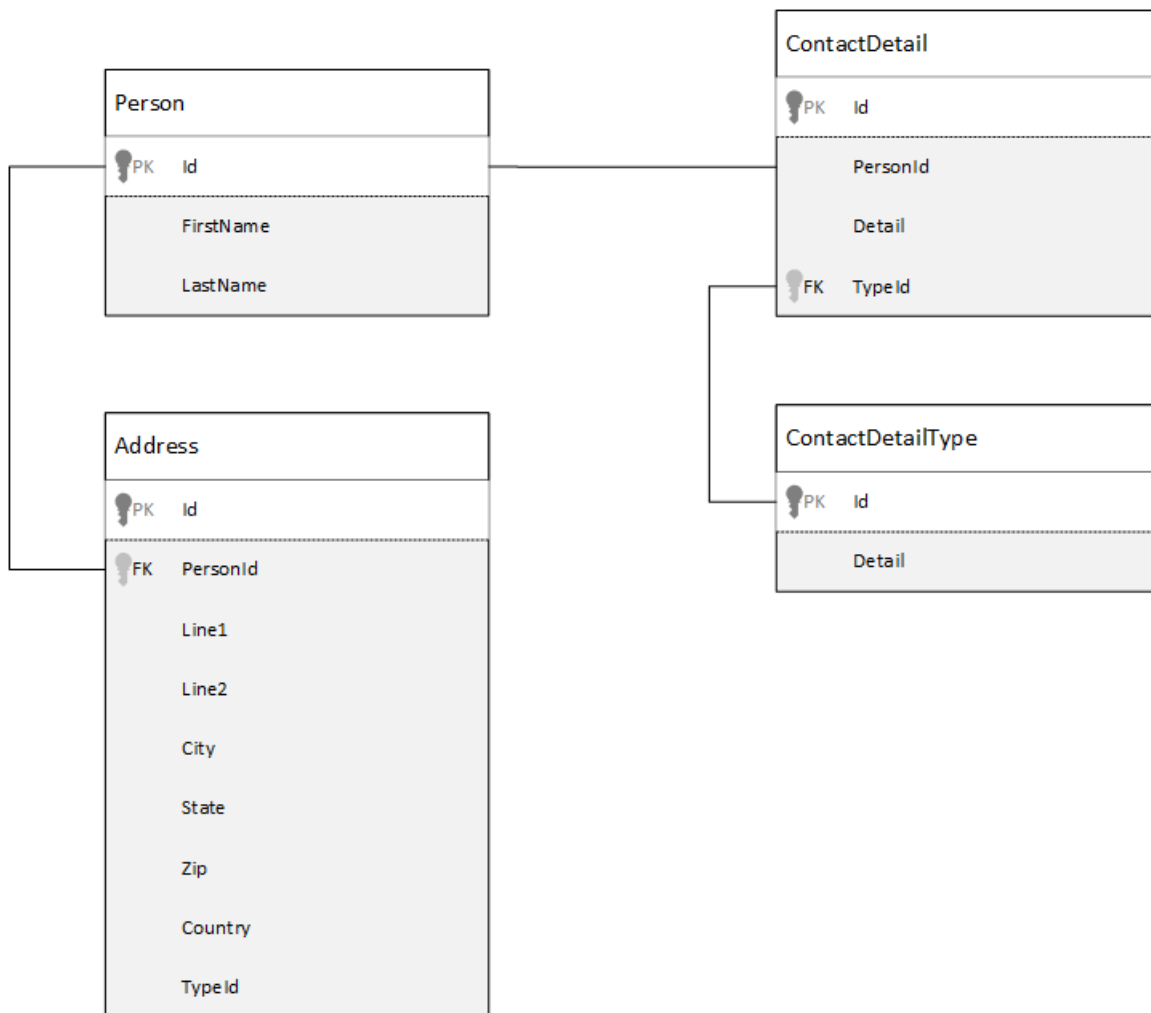
After reading this article, you will be able to answer the following questions:

- How should I think about a document in a document database?
- What is data modeling and why should I care?
- How is modeling data in a document database different to a relational database?
- How do I express data relationships in a non-relational database?
- When do I embed data and when do I link to data?

Embedding data

When you start modeling data in a document store, such as DocumentDB, try to treat your entities as **self-contained documents** represented in JSON.

Before we dive in too much further, let us take a few steps back and have a look at how we might model something in a relational database, a subject many of us are already familiar with. The following example shows how a person might be stored in a relational database.



When working with relational databases, we've been taught for years to normalize, normalize, normalize.

Normalizing your data typically involves taking an entity, such as a person, and breaking it down in to discrete pieces of data. In the example above, a person can have multiple contact detail records as well as multiple address records. We even go one step further and break down contact details by further extracting common fields like a type. Same for address, each record here has a type like *Home* or *Business*

The guiding premise when normalizing data is to **avoid storing redundant data** on each record and rather refer to data. In this example, to read a person, with all their contact details and addresses, you need to use JOINS to effectively aggregate your data at run time.

```
SELECT p.FirstName, p.LastName, a.City, cd.Detail
FROM Person p
JOIN ContactDetail cd ON cd.PersonId = p.Id
JOIN ContactDetailType cdt ON cdt.Id = cd.TypeId
JOIN Address a ON a.PersonId = p.Id
```

Updating a single person with their contact details and addresses requires write operations across many individual tables.

Now let's take a look at how we would model the same data as a self-contained entity in a document database.

```
{
  "id": "1",
  "firstName": "Thomas",
  "lastName": "Andersen",
  "addresses": [
    {
      "line1": "100 Some Street",
      "line2": "Unit 1",
      "city": "Seattle",
      "state": "WA",
      "zip": 98012
    }
  ],
  "contactDetails": [
    { "email": "thomas@andersen.com" },
    { "phone": "+1 555 555-5555", "extension": 5555 }
  ]
}
```

Using the approach above we have now **denormalized** the person record where we **embedded** all the information relating to this person, such as their contact details and addresses, in to a single JSON document. In addition, because we're not confined to a fixed schema we have the flexibility to do things like having contact details of different shapes entirely.

Retrieving a complete person record from the database is now a single read operation against a single collection and for a single document. Updating a person record, with their contact details and addresses, is also a single write operation against a single document.

By denormalizing data, your application may need to issue fewer queries and updates to complete common operations.

When to embed

In general, use embedded data models when:

- There are **contains** relationships between entities.
- There are **one-to-few** relationships between entities.
- There is embedded data that **changes infrequently**.
- There is embedded data won't grow **without bound**.
- There is embedded data that is **integral** to data in a document.

NOTE

Typically denormalized data models provide better **read** performance.

When not to embed

While the rule of thumb in a document database is to denormalize everything and embed all data in to a single document, this can lead to some situations that should be avoided.

Take this JSON snippet.

```
{
  "id": "1",
  "name": "What's new in the coolest Cloud",
  "summary": "A blog post by someone real famous",
  "comments": [
    {"id": 1, "author": "anon", "comment": "something useful, I'm sure"},
    {"id": 2, "author": "bob", "comment": "wisdom from the interwebs"},
    ...
    {"id": 100001, "author": "jane", "comment": "and on we go ..."},
    ...
    {"id": 1000000001, "author": "angry", "comment": "blah angry blah angry"},
    ...
    {"id": ∞ + 1, "author": "bored", "comment": "oh man, will this ever end?"},
  ]
}
```

This might be what a post entity with embedded comments would look like if we were modeling a typical blog, or CMS, system. The problem with this example is that the comments array is **unbounded**, meaning that there is no (practical) limit to the number of comments any single post can have. This will become a problem as the size of the document could grow significantly.

TIP

Documents in DocumentDB have a maximum size. For more on this refer to [DocumentDB limits](#).

As the size of the document grows the ability to transmit the data over the wire as well as reading and updating the document, at scale, will be impacted.

In this case it would be better to consider the following model.

```
Post document:
{
  "id": "1",
  "name": "What's new in the coolest Cloud",
  "summary": "A blog post by someone real famous",
  "recentComments": [
    {"id": 1, "author": "anon", "comment": "something useful, I'm sure"},
    {"id": 2, "author": "bob", "comment": "wisdom from the interwebs"},
    {"id": 3, "author": "jane", "comment": "...."}
  ]
}

Comment documents:
{
  "postId": "1"
  "comments": [
    {"id": 4, "author": "anon", "comment": "more goodness"},
    {"id": 5, "author": "bob", "comment": "tails from the field"},
    ...
    {"id": 99, "author": "angry", "comment": "blah angry blah angry"}
  ]
},
{
  "postId": "1"
  "comments": [
    {"id": 100, "author": "anon", "comment": "yet more"},
    ...
    {"id": 199, "author": "bored", "comment": "will this ever end?"}
  ]
}
```

This model has the three most recent comments embedded on the post itself, which is an array with a fixed bound

this time. The other comments are grouped in to batches of 100 comments and stored in separate documents. The size of the batch was chosen as 100 because our fictitious application allows the user to load 100 comments at a time.

Another case where embedding data is not a good idea is when the embedded data is used often across documents and will change frequently.

Take this JSON snippet.

```
{
  "id": "1",
  "firstName": "Thomas",
  "lastName": "Andersen",
  "holdings": [
    {
      "numberHeld": 100,
      "stock": { "symbol": "zaza", "open": 1, "high": 2, "low": 0.5 }
    },
    {
      "numberHeld": 50,
      "stock": { "symbol": "xcxc", "open": 89, "high": 93.24, "low": 88.87 }
    }
  ]
}
```

This could represent a person's stock portfolio. We have chosen to embed the stock information in to each portfolio document. In an environment where related data is changing frequently, like a stock trading application, embedding data that changes frequently is going to mean that you are constantly updating each portfolio document every time a stock is traded.

Stock *zaza* may be traded many hundreds of times in a single day and thousands of users could have *zaza* on their portfolio. With a data model like the above we would have to update many thousands of portfolio documents many times every day leading to a system that won't scale very well.

Referencing data

So, embedding data works nicely for many cases but it is clear that there are scenarios when denormalizing your data will cause more problems than it is worth. So what do we do now?

Relational databases are not the only place where you can create relationships between entities. In a document database you can have information in one document that actually relates to data in other documents. Now, I am not advocating for even one minute that we build systems that would be better suited to a relational database in DocumentDB, or any other document database, but simple relationships are fine and can be very useful.

In the JSON below we chose to use the example of a stock portfolio from earlier but this time we refer to the stock item on the portfolio instead of embedding it. This way, when the stock item changes frequently throughout the day the only document that needs to be updated is the single stock document.

```

Person document:
{
  "id": "1",
  "firstName": "Thomas",
  "lastName": "Andersen",
  "holdings": [
    { "numberHeld": 100, "stockId": 1},
    { "numberHeld": 50, "stockId": 2}
  ]
}

Stock documents:
{
  "id": "1",
  "symbol": "zaza",
  "open": 1,
  "high": 2,
  "low": 0.5,
  "vol": 11970000,
  "mkt-cap": 42000000,
  "pe": 5.89
},
{
  "id": "2",
  "symbol": "xcxc",
  "open": 89,
  "high": 93.24,
  "low": 88.87,
  "vol": 2970200,
  "mkt-cap": 1005000,
  "pe": 75.82
}

```

An immediate downside to this approach though is if your application is required to show information about each stock that is held when displaying a person's portfolio; in this case you would need to make multiple trips to the database to load the information for each stock document. Here we've made a decision to improve the efficiency of write operations, which happen frequently throughout the day, but in turn compromised on the read operations that potentially have less impact on the performance of this particular system.

NOTE

Normalized data models can require more round trips to the server.

What about foreign keys?

Because there is currently no concept of a constraint, foreign-key or otherwise, any inter-document relationships that you have in documents are effectively "weak links" and will not be verified by the database itself. If you want to ensure that the data a document is referring to actually exists, then you need to do this in your application, or through the use of server-side triggers or stored procedures on DocumentDB.

When to reference

In general, use normalized data models when:

- Representing **one-to-many** relationships.
- Representing **many-to-many** relationships.
- Related data **changes frequently**.
- Referenced data could be **unbounded**.

NOTE

Typically normalizing provides better **write** performance.

Where do I put the relationship?

The growth of the relationship will help determine in which document to store the reference.

If we look at the JSON below that models publishers and books.

```
Publisher document:
{
  "id": "mspress",
  "name": "Microsoft Press",
  "books": [ 1, 2, 3, ..., 100, ..., 1000]
}

Book documents:
{"id": "1", "name": "DocumentDB 101" }
{"id": "2", "name": "DocumentDB for RDBMS Users" }
{"id": "3", "name": "Taking over the world one JSON doc at a time" }
...
{"id": "100", "name": "Learn about Azure DocumentDB" }
...
{"id": "1000", "name": "Deep Dive in to DocumentDB" }
```

If the number of the books per publisher is small with limited growth, then storing the book reference inside the publisher document may be useful. However, if the number of books per publisher is unbounded, then this data model would lead to mutable, growing arrays, as in the example publisher document above.

Switching things around a bit would result in a model that still represents the same data but now avoids these large mutable collections.

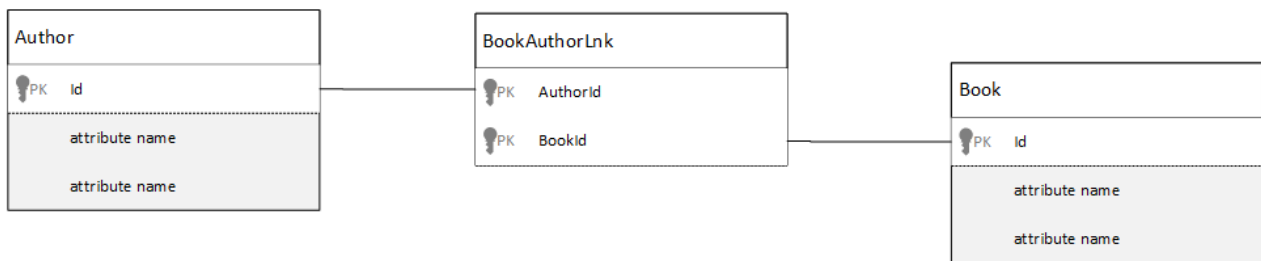
```
Publisher document:
{
  "id": "mspress",
  "name": "Microsoft Press"
}

Book documents:
{"id": "1", "name": "DocumentDB 101", "pub-id": "mspress"}
{"id": "2", "name": "DocumentDB for RDBMS Users", "pub-id": "mspress"}
{"id": "3", "name": "Taking over the world one JSON doc at a time"}
...
{"id": "100", "name": "Learn about Azure DocumentDB", "pub-id": "mspress"}
...
{"id": "1000", "name": "Deep Dive in to DocumentDB", "pub-id": "mspress"}
```

In the above example, we have dropped the unbounded collection on the publisher document. Instead we just have a reference to the publisher on each book document.

How do I model many:many relationships?

In a relational database *many:many* relationships are often modeled with join tables, which just join records from other tables together.



You might be tempted to replicate the same thing using documents and produce a data model that looks similar to the following.

```
Author documents:
{"id": "a1", "name": "Thomas Andersen" }
{"id": "a2", "name": "William Wakefield" }

Book documents:
{"id": "b1", "name": "DocumentDB 101" }
{"id": "b2", "name": "DocumentDB for RDBMS Users" }
{"id": "b3", "name": "Taking over the world one JSON doc at a time" }
{"id": "b4", "name": "Learn about Azure DocumentDB" }
{"id": "b5", "name": "Deep Dive in to DocumentDB" }

Joining documents:
{"authorId": "a1", "bookId": "b1" }
{"authorId": "a2", "bookId": "b1" }
{"authorId": "a1", "bookId": "b2" }
{"authorId": "a1", "bookId": "b3" }
```

This would work. However, loading either an author with their books, or loading a book with its author, would always require at least two additional queries against the database. One query to the joining document and then another query to fetch the actual document being joined.

If all this join table is doing is gluing together two pieces of data, then why not drop it completely? Consider the following.

```
Author documents:
{"id": "a1", "name": "Thomas Andersen", "books": ["b1", "b2", "b3"]}
{"id": "a2", "name": "William Wakefield", "books": ["b1", "b4"]}

Book documents:
{"id": "b1", "name": "DocumentDB 101", "authors": ["a1", "a2"]}
{"id": "b2", "name": "DocumentDB for RDBMS Users", "authors": ["a1"]}
{"id": "b3", "name": "Learn about Azure DocumentDB", "authors": ["a1"]}
{"id": "b4", "name": "Deep Dive in to DocumentDB", "authors": ["a2"]}
```

Now, if I had an author, I immediately know which books they have written, and conversely if I had a book document loaded I would know the ids of the author(s). This saves that intermediary query against the join table reducing the number of server round trips your application has to make.

Hybrid data models

We've now looked embedding (or denormalizing) and referencing (or normalizing) data, each have their upsides and each have compromises as we have seen.

It doesn't always have to be either or, don't be scared to mix things up a little.

Based on your application's specific usage patterns and workloads there may be cases where mixing embedded and referenced data makes sense and could lead to simpler application logic with fewer server round trips while still maintaining a good level of performance.

Consider the following JSON.

```
Author documents:
{
  "id": "a1",
  "firstName": "Thomas",
  "lastName": "Andersen",
  "countOfBooks": 3,
  "books": ["b1", "b2", "b3"],
  "images": [
    {"thumbnail": "http://....png"}
    {"profile": "http://....png"}
    {"large": "http://....png"}
  ]
},
{
  "id": "a2",
  "firstName": "William",
  "lastName": "Wakefield",
  "countOfBooks": 1,
  "books": ["b1"],
  "images": [
    {"thumbnail": "http://....png"}
  ]
}

Book documents:
{
  "id": "b1",
  "name": "DocumentDB 101",
  "authors": [
    {"id": "a1", "name": "Thomas Andersen", "thumbnailUrl": "http://....png"},
    {"id": "a2", "name": "William Wakefield", "thumbnailUrl": "http://....png"}
  ]
},
{
  "id": "b2",
  "name": "DocumentDB for RDBMS Users",
  "authors": [
    {"id": "a1", "name": "Thomas Andersen", "thumbnailUrl": "http://....png"},
  ]
}
```

Here we've (mostly) followed the embedded model, where data from other entities are embedded in the top-level document, but other data is referenced.

If you look at the book document, we can see a few interesting fields when we look at the array of authors. There is an *id* field which is the field we use to refer back to an author document, standard practice in a normalized model, but then we also have *name* and *thumbnailUrl*. We could've just stuck with *id* and left the application to get any additional information it needed from the respective author document using the "link", but because our application displays the author's name and a thumbnail picture with every book displayed we can save a round trip to the server per book in a list by denormalizing **some** data from the author.

Sure, if the author's name changed or they wanted to update their photo we'd have to go and update every book they ever published but for our application, based on the assumption that authors don't change their names very often, this is an acceptable design decision.

In the example there are **pre-calculated aggregates** values to save expensive processing on a read operation. In the example, some of the data embedded in the author document is data that is calculated at run-time. Every time a new book is published, a book document is created **and** the *countOfBooks* field is set to a calculated value based on the number of book documents that exist for a particular author. This optimization would be good in read heavy systems where we can afford to do computations on writes in order to optimize reads.

The ability to have a model with pre-calculated fields is made possible because DocumentDB supports **multi-document transactions**. Many NoSQL stores cannot do transactions across documents and therefore advocate design decisions, such as "always embed everything", due to this limitation. With DocumentDB, you can use server-side triggers, or stored procedures, that insert books and update authors all within an ACID transaction. Now you don't **have** to embed everything in to one document just to be sure that your data remains consistent.

Next steps

The biggest takeaways from this article is to understand that data modeling in a schema-free world is just as important as ever.

Just as there is no single way to represent a piece of data on a screen, there is no single way to model your data. You need to understand your application and how it will produce, consume, and process the data. Then, by applying some of the guidelines presented here you can set about creating a model that addresses the immediate needs of your application. When your applications need to change, you can leverage the flexibility of a schema-free database to embrace that change and evolve your data model easily.

To learn more about Azure DocumentDB, refer to the service's [documentation](#) page.

To learn about tuning indexes in Azure DocumentDB, refer to the article on [indexing policies](#).

To understand how to shard your data across multiple partitions, refer to [Partitioning Data in DocumentDB](#).

And finally, for guidance on modeling data and sharding for multi-tenant applications, consult [Scaling a Multi-Tenant Application with Azure DocumentDB](#).

Working with Geospatial data in Azure DocumentDB

11/22/2016 • 11 min to read • [Edit on GitHub](#)

Contributors

[arramac](#) • [mimig](#) • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [Ross McAllister](#) • [Dave](#) • [v-aljenk](#) • [jastru](#)

This article is an introduction to the geospatial functionality in [Azure 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](#) 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](#). 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 ]
}
```

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 ]
  ]
}
```

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](#). 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](#).

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](#).

Usage	Description
-------	-------------

ST_DISTANCE (spatial_expr, spatial_expr)	Returns the distance between the two GeoJSON Point, Polygon, or LineString expressions.
ST_WITHIN (spatial_expr, spatial_expr)	Returns a Boolean expression indicating whether the first GeoJSON object (Point, Polygon, or LineString) is within the second GeoJSON object (Point, Polygon, or LineString).
ST_INTERSECTS (spatial_expr, spatial_expr)	Returns a Boolean expression indicating whether the two specified GeoJSON objects (Point, Polygon, or LineString) intersect.
ST_ISVALID	Returns a Boolean value indicating whether the specified GeoJSON Point, Polygon, or LineString expression is valid.
ST_ISVALIDDETAILED	Returns a JSON value containing a Boolean value if the specified GeoJSON Point, Polygon, or LineString 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](#) 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.

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",
}]
```

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 spatial type is invalid.

DocumentDB also supports performing inverse queries, i.e. you can index Polygons or lines in DocumentDB, then query for the areas that contain a specified point. This pattern is commonly used in logistics to identify e.g. when a truck enters or leaves a designated area.

Query

```
SELECT *
FROM Areas a
WHERE ST_WITHIN({'type': 'Point', 'coordinates':[31.9, -4.8]}, a.location)
```

Results

```
[{
  "id": "MyDesignatedLocation",
  "location": {
    "type": "Polygon",
    "coordinates": [[[31.8, -5], [32, -5], [32, -4.7], [31.8, -4.7], [31.8, -5]]]
  }
}]
```

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 provides 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](#) 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 natively understands spatial data (points, Polygons and lines) 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.

NOTE

DocumentDB supports automatic indexing of Points, Polygons, and LineStrings

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 for points and Polygons

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

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](#)
- Get hands on with geospatial querying at the [DocumentDB Query Playground](#)
- Learn more about [DocumentDB Query](#)
- Learn more about [DocumentDB Indexing Policies](#)

Developing with multi-region DocumentDB accounts

11/15/2016 • 4 min to read • [Edit on GitHub](#)

Contributors

Kirat Pandya • Theano Petersen • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • arramac • mimig • Matt Scully

NOTE

Global distribution of DocumentDB databases is generally available and automatically enabled for any newly created DocumentDB accounts. We are working to enable global distribution on all existing accounts, but in the interim, if you want global distribution enabled on your account, please [contact support](#) and we'll enable it for you now.

In order to take advantage of [global distribution](#), client applications can specify the ordered preference list of regions to be used to perform document operations. This can be done by setting the connection policy. Based on the Azure DocumentDB account configuration, current regional availability and the preference list specified, the most optimal endpoint will be chosen by the SDK to perform write and read operations.

This preference list is specified when initializing a connection using the DocumentDB client SDKs. The SDKs accept an optional parameter "PreferredLocations" that is an ordered list of Azure regions.

The SDK will automatically send all writes to the current write region.

All reads will be sent to the first available region in the PreferredLocations list. If the request fails, the client will fail down the list to the next region, and so on.

The client SDKs will only attempt to read from the regions specified in PreferredLocations. So, for example, if the Database Account is available in three regions, but the client only specifies two of the non-write regions for PreferredLocations, then no reads will be served out of the write region, even in the case of failover.

The application can verify the current write endpoint and read endpoint chosen by the SDK by checking two properties, WriteEndpoint and ReadEndpoint, available in SDK version 1.8 and above.

If the PreferredLocations property is not set, all requests will be served from the current write region.

.NET SDK

The SDK can be used without any code changes. In this case, the SDK automatically directs both reads and writes to the current write region.

In version 1.8 and later of the .NET SDK, the ConnectionPolicy parameter for the DocumentClient constructor has a property called Microsoft.Azure.Documents.ConnectionPolicy.PreferredLocations. This property is of type Collection `<string>` and should contain a list of region names. The string values are formatted per the Region Name column on the [Azure Regions](#) page, with no spaces before or after the first and last character respectively.

The current write and read endpoints are available in DocumentClient.WriteEndpoint and DocumentClient.ReadEndpoint respectively.

NOTE

The URLs for the endpoints should not be considered as long-lived constants. The service may update these at any point. The SDK handles this change automatically.

```
// Getting endpoints from application settings or other configuration location
Uri accountEndPoint = new Uri(Properties.Settings.Default.GlobalDatabaseUri);
string accountKey = Properties.Settings.Default.GlobalDatabaseKey;

//Setting read region selection preference
connectionPolicy.PreferredLocations.Add(LocationNames.WestUS); // first preference
connectionPolicy.PreferredLocations.Add(LocationNames.EastUS); // second preference
connectionPolicy.PreferredLocations.Add(LocationNames.NorthEurope); // third preference

// initialize connection
DocumentClient docClient = new DocumentClient(
    accountEndPoint,
    accountKey,
    connectionPolicy);

// connect to DocDB
await docClient.OpenAsync().ConfigureAwait(false);
```

NodeJS, JavaScript, and Python SDKs

The SDK can be used without any code changes. In this case, the SDK will automatically direct both reads and writes to the current write region.

In version 1.8 and later of each SDK, the ConnectionPolicy parameter for the DocumentClient constructor a new property called DocumentClient.ConnectionPolicy.PreferredLocations. This is parameter is an array of strings that takes a list of region names. The names are formatted per the Region Name column in the [Azure Regions](#) page. You can also use the predefined constants in the convenience object AzureDocuments.Regions

The current write and read endpoints are available in DocumentClient.getWriteEndpoint and DocumentClient.getReadEndpoint respectively.

NOTE

The URLs for the endpoints should not be considered as long-lived constants. The service may update these at any point. The SDK will handle this change automatically.

Below is a code example for NodeJS/Javascript. Python and Java will follow the same pattern.

```
// Creating a ConnectionPolicy object
var connectionPolicy = new DocumentBase.ConnectionPolicy();

// Setting read region selection preference, in the following order -
// 1 - West US
// 2 - East US
// 3 - North Europe
connectionPolicy.PreferredLocations = ['West US', 'East US', 'North Europe'];

// initialize the connection
var client = new DocumentDBClient(host, { masterKey: masterKey }, connectionPolicy);
```

REST

Once a database account has been made available in multiple regions, clients can query its availability by performing a GET request on the following URI.

```
https://{databaseaccount}.documents.azure.com/
```

The service will return a list of regions and their corresponding DocumentDB endpoint URIs for the replicas. The current write region will be indicated in the response. The client can then select the appropriate endpoint for all further REST API requests as follows.

Example response

```
{
  "_dbs": "//dbs/",
  "media": "//media/",
  "writableLocations": [
    {
      "Name": "West US",
      "DatabaseAccountEndpoint": "https://globaldbexample-westus.documents.azure.com:443/"
    }
  ],
  "readableLocations": [
    {
      "Name": "East US",
      "DatabaseAccountEndpoint": "https://globaldbexample-eastus.documents.azure.com:443/"
    }
  ],
  "MaxMediaStorageUsageInMB": 2048,
  "MediaStorageUsageInMB": 0,
  "ConsistencyPolicy": {
    "defaultConsistencyLevel": "Session",
    "maxStalenessPrefix": 100,
    "maxIntervalInSeconds": 5
  },
  "addresses": "//addresses/",
  "id": "globaldbexample",
  "_rid": "globaldbexample.documents.azure.com",
  "_self": "",
  "_ts": 0,
  "_etag": null
}
```

- All PUT, POST and DELETE requests must go to the indicated write URI
- All GETs and other read-only requests (for example queries) may go to any endpoint of the client's choice

Write requests to read-only regions will fail with HTTP error code 403 ("Forbidden").

If the write region changes after the client's initial discovery phase, subsequent writes to the previous write region will fail with HTTP error code 403 ("Forbidden"). The client should then GET the list of regions again to get the updated write region.

Next steps

Learn more about the distributing data globally with DocumentDB in the following articles:

- [Distribute data globally with DocumentDB](#)
- [Consistency levels](#)
- [How throughput works with multiple regions](#)
- [Add regions using the Azure portal](#)

Expire data in DocumentDB collections automatically with time to live

11/22/2016 • 6 min to read • [Edit on GitHub](#)

Contributors

Kirat Pandya • mimig • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

Applications can produce and store vast amounts of data. Some of this data, like machine generated event data, logs, and user session information is only useful for a finite period of time. Once the data becomes surplus to the needs of the application it is safe to purge this data and reduce the storage needs of an application.

With "time to live" or TTL, Microsoft Azure DocumentDB provides the ability to have documents automatically purged from the database after a period of time. The default time to live can be set at the collection level, and overridden on a per-document basis. Once TTL is set, either as a collection default or at a document level, DocumentDB will automatically remove documents that exist after that period of time, in seconds, since they were last modified.

Time to live in DocumentDB uses an offset against when the document was last modified. To do this it uses the `_ts` field which exists on every document. The `_ts` field is a unix-style epoch timestamp representing the date and time. The `_ts` field is updated every time a document is modified.

TTL behavior

The TTL feature is controlled by TTL properties at two levels - the collection level and the document level. The values are set in seconds and are treated as a delta from the `_ts` that the document was last modified at.

1. DefaultTTL for the collection

- If missing (or set to null), documents are not deleted automatically.
- If present and the value is "-1" = infinite – documents don't expire by default
- If present and the value is some number ("n") – documents expire "n" seconds after last modification

2. TTL for the documents:

- Property is applicable only if DefaultTTL is present for the parent collection.
- Overrides the DefaultTTL value for the parent collection.

As soon as the document has expired ($t_{tl} + _ts \geq \text{current server time}$), the document is marked as "expired". No operation will be allowed on these documents after this time and they will be excluded from the results of any queries performed. The documents are physically deleted in the system, and are deleted in the background opportunistically at a later time. This does not consume any [Request Units \(RUs\)](#) from the collection budget.

The above logic can be shown in the following matrix:

	DEFAULTTTL MISSING/NOT SET ON THE COLLECTION	DEFAULTTTL = -1 ON COLLECTION	DEFAULTTTL = "N" ON COLLECTION
TTL Missing on document	Nothing to override at document level since both the document and collection have no concept of TTL.	No documents in this collection will expire.	The documents in this collection will expire when interval n elapses.

	DEFAULTTTL MISSING/NOT SET ON THE COLLECTION	DEFAULTTTL = -1 ON COLLECTION	DEFAULTTTL = "N" ON COLLECTION
TTL = -1 on document	Nothing to override at the document level since the collection doesn't define the DefaultTTL property that a document can override. TTL on a document is un-interpreted by the system.	No documents in this collection will expire.	The document with TTL=-1 in this collection will never expire. All other documents will expire after "n" interval.
TTL = n on document	Nothing to override at the document level. TTL on a document is un-interpreted by the system.	The document with TTL = n will expire after interval n, in seconds. Other documents will inherit interval of -1 and never expire.	The document with TTL = n will expire after interval n, in seconds. Other documents will inherit "n" interval from the collection.

Configuring TTL

By default, time to live is disabled by default in all DocumentDB collections and on all documents.

Enabling TTL

To enable TTL on a collection, or the documents within a collection, you need to set the DefaultTTL property of a collection to either -1 or a non-zero positive number. Setting the DefaultTTL to -1 means that by default all documents in the collection will live forever but the DocumentDB service should monitor this collection for documents that have overridden this default.

Configuring default TTL on a collection

You are able to configure a default time to live at a collection level.

To set the TTL on a collection, you need to provide a non-zero positive number that indicates the period, in seconds, to expire all documents in the collection after the last modified timestamp of the document (`_ts`).

Or, you can set the default to -1, which implies that all documents inserted in to the collection will live indefinitely by default.

Setting TTL on a document

In addition to setting a default TTL on a collection you can set specific TTL at a document level. Doing this will override the default of the collection.

To set the TTL on a document, you need to provide a non-zero positive number which indicates the period, in seconds, to expire the document after the last modified timestamp of the document (`_ts`).

To set this expiry offset, set the TTL field on the document.

If a document has no TTL field, then the default of the collection will apply.

If TTL is disabled at the collection level, the TTL field on the document will be ignored until TTL is enabled again on the collection.

Extending TTL on an existing document

You can reset the TTL on a document by doing any write operation on the document. Doing this will set the `_ts` to the current time, and the countdown to the document expiry, as set by the `ttl`, will begin again.

If you wish to change the ttl of a document, you can update the field as you can do with any other settable field.

Removing TTL from a document

If a TTL has been set on a document and you no longer want that document to expire, then you can retrieve the document, remove the TTL field and replace the document on the server.

When the TTL field is removed from the document, the default of the collection will be applied.

To stop a document from expiring and not inherit from the collection then you need to set the TTL value to -1.

Disabling TTL

To disable TTL entirely on a collection and stop the background process from looking for expired documents the DefaultTTL property on the collection should be deleted.

Deleting this property is different from setting it to -1. Setting to -1 means new documents added to the collection will live forever but you can override this on specific documents in the collection.

Removing this property entirely from the collection means that no documents will expire, even if there are documents that have explicitly overridden a previous default.

FAQ

What will TTL cost me?

There is no additional cost to setting a TTL on a document.

How long will it take to delete my document once the TTL is up?

The documents are expired immediately once the TTL is up, and will not be accessible via CRUD or query APIs.

Will TTL on a document have any impact on RU charges?

No, there will be no impact on RU charges for deletions of expired documents via TTL in DocumentDB.

Does the TTL feature only apply to entire documents, or can I expire individual document property values?

TTL applies to the entire document. If you would like to expire just a portion of a document, then it is recommended that you extract the portion from the main document in to a separate "linked" document and then use TTL on that extracted document.

Does the TTL feature have any specific indexing requirements?

Yes. The collection must have [indexing policy set](#) to either Consistent or Lazy. Trying to set DefaultTTL on a collection with indexing set to None will result in an error, as will trying to turn off indexing on a collection that has a DefaultTTL already set.

Next steps

To learn more about Azure DocumentDB, refer to the service [documentation](#) page.

DocumentDB indexing policies

11/15/2016 • 18 min to read • [Edit on GitHub](#)

Contributors

arramac • mimig • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ross McAllister • Jennifer Hubbard

While many customers are happy to let Azure DocumentDB automatically handle [all aspects of indexing](#), DocumentDB also supports specifying a custom **indexing policy** for collections during creation. Indexing policies in DocumentDB are more flexible and powerful than secondary indexes offered in other database platforms, because they let you design and customize the shape of the index without sacrificing schema flexibility. To learn how indexing works within DocumentDB, you must understand that by managing indexing policy, you can make fine-grained tradeoffs between index storage overhead, write and query throughput, and query consistency.

In this article, we take a close look at DocumentDB indexing policies, how you can customize indexing policy, and the associated trade-offs.

After reading this article, you'll be able to answer the following questions:

- How can I override the properties to include or exclude from indexing?
- How can I configure the index for eventual updates?
- How can I configure indexing to perform Order By or range queries?
- How do I make changes to a collection's indexing policy?
- How do I compare storage and performance of different indexing policies?

Customizing the indexing policy of a collection

Developers can customize the trade-offs between storage, write/query performance, and query consistency, by overriding the default indexing policy on a DocumentDB collection and configuring the following aspects.

- **Including/Excluding documents and paths to/from index.** Developers can choose certain documents to be excluded or included in the index at the time of inserting or replacing them to the collection. Developers can also choose to include or exclude certain JSON properties a.k.a. paths (including wildcard patterns) to be indexed across documents which are included in an index.
- **Configuring Various Index Types.** For each of the included paths, developers can also specify the type of index they require over a collection based on their data and expected query workload and the numeric/string "precision" for each path.
- **Configuring Index Update Modes.** DocumentDB supports three indexing modes which can be configured via the indexing policy on a DocumentDB collection: Consistent, Lazy and None.

The following .NET code snippet shows how to set a custom indexing policy during the creation of a collection. Here we set the policy with Range index for strings and numbers at the maximum precision. This policy lets us execute Order By queries against strings.

```
DocumentCollection collection = new DocumentCollection { Id = "myCollection" };

collection.IndexingPolicy = new IndexingPolicy(new RangeIndex(DataType.String) { Precision = -1 });
collection.IndexingPolicy.IndexingMode = IndexingMode.Consistent;

await client.CreateDocumentCollectionAsync(UriFactory.CreateDatabaseUri("db"), collection);
```

NOTE

The JSON schema for indexing policy was changed with the release of REST API version 2015-06-03 to support Range indexes against strings. .NET SDK 1.2.0 and Java, Python, and Node.js SDKs 1.1.0 support the new policy schema. Older SDKs use the REST API version 2015-04-08 and support the older schema of Indexing Policy.

By default, DocumentDB indexes all string properties within documents consistently with a Hash index, and numeric properties with a Range index.

Database indexing modes

DocumentDB supports three indexing modes which can be configured via the indexing policy on a DocumentDB collection – Consistent, Lazy and None.

Consistent: If a DocumentDB collection's policy is designated as "consistent", the queries on a given DocumentDB collection follow the same consistency level as specified for the point-reads (i.e. strong, bounded-staleness, session or eventual). The index is updated synchronously as part of the document update (i.e. insert, replace, update, and delete of a document in a DocumentDB collection). Consistent indexing supports consistent queries at the cost of possible reduction in write throughput. This reduction is a function of the unique paths that need to be indexed and the "consistency level". Consistent indexing mode is designed for "write quickly, query immediately" workloads.

Lazy: To allow maximum document ingestion throughput, a DocumentDB collection can be configured with lazy consistency; meaning queries are eventually consistent. The index is updated asynchronously when a DocumentDB collection is quiescent i.e. when the collection's throughput capacity is not fully utilized to serve user requests. For "ingest now, query later" workloads requiring unhindered document ingestion, "lazy" indexing mode may be suitable.

None: A collection marked with index mode of "None" has no index associated with it. This is commonly used if DocumentDB is utilized as a key-value storage and documents are accessed only by their ID property.

NOTE

Configuring the indexing policy with "None" has the side effect of dropping any existing index. Use this if your access patterns are only require "id" and/or "self-link".

The following sample show how create a DocumentDB collection using the .NET SDK with consistent automatic indexing on all document insertions.

The following table shows the consistency for queries based on the indexing mode (Consistent and Lazy) configured for the collection and the consistency level specified for the query request. This applies to queries made using any interface - REST API, SDKs or from within stored procedures and triggers.

CONSISTENCY	INDEXING MODE: CONSISTENT	INDEXING MODE: LAZY
Strong	Strong	Eventual
Bounded Staleness	Bounded Staleness	Eventual
Session	Session	Eventual
Eventual	Eventual	Eventual

DocumentDB returns an error for queries made on collections with None indexing mode. Queries can still be executed as scans via the explicit `x-ms-documentdb-enable-scan` header in the REST API or the `EnableScanInQuery`

request option using the .NET SDK. Some query features like ORDER BY are not supported as scans with `EnableScanInQuery` .

The following table shows the consistency for queries based on the indexing mode (Consistent, Lazy, and None) when `EnableScanInQuery` is specified.

CONSISTENCY	INDEXING MODE: CONSISTENT	INDEXING MODE: LAZY	INDEXING MODE: NONE
Strong	Strong	Eventual	Strong
Bounded Staleness	Bounded Staleness	Eventual	Bounded Staleness
Session	Session	Eventual	Session
Eventual	Eventual	Eventual	Eventual

The following code sample show how create a DocumentDB collection using the .NET SDK with consistent indexing on all document insertions.

```
// Default collection creates a hash index for all string and numeric
// fields. Hash indexes are compact and offer efficient
// performance for equality queries.

var collection = new DocumentCollection { Id ="defaultCollection" };

collection.IndexingPolicy.IndexingMode = IndexingMode.Consistent;

collection = await client.CreateDocumentCollectionAsync(UriFactory.CreateDatabaseUri("mydb"), collection);
```

Index paths

DocumentDB models JSON documents and the index as trees, and allows you to tune to policies for paths within the tree. You can find more details in this [introduction to DocumentDB indexing](#). Within documents, you can choose which paths must be included or excluded from indexing. This can offer improved write performance and lower index storage for scenarios when the query patterns are known beforehand.

Index paths start with the root (/) and typically end with the ? wildcard operator, denoting that there are multiple possible values for the prefix. For example, to serve `SELECT * FROM Families F WHERE F.familyName = "Andersen"`, you must include an index path for `/familyName/?` in the collection’s index policy.

Index paths can also use the * wildcard operator to specify the behavior for paths recursively under the prefix. For example, `/payload/*` can be used to exclude everything under the payload property from indexing.

Here are the common patterns for specifying index paths:

PATH	DESCRIPTION/USE CASE
/	Default path for collection. Recursive and applies to whole document tree.

PATH	DESCRIPTION/USE CASE
/prop/?	<p>Index path required to serve queries like the following (with Hash or Range types respectively):</p> <pre>SELECT FROM collection c WHERE c.prop = "value"</pre> <pre>SELECT FROM collection c WHERE c.prop > 5</pre> <pre>SELECT FROM collection c ORDER BY c.prop</pre>
/prop/	<p>Index path for all paths under the specified label. Works with the following queries</p> <pre>SELECT FROM collection c WHERE c.prop = "value"</pre> <pre>SELECT FROM collection c WHERE c.prop.subprop > 5</pre> <pre>SELECT FROM collection c WHERE c.prop.subprop.nextprop = "value"</pre> <pre>SELECT FROM collection c ORDER BY c.prop</pre>
/props/[]/?	<p>Index path required to serve iteration and JOIN queries against arrays of scalars like ["a", "b", "c"]:</p> <pre>SELECT tag FROM tag IN collection.props WHERE tag = "value"</pre> <pre>SELECT tag FROM collection c JOIN tag IN c.props WHERE tag > 5</pre>
/props/[]/subprop/?	<p>Index path required to serve iteration and JOIN queries against arrays of objects like [{subprop: "a"}, {subprop: "b"}]:</p> <pre>SELECT tag FROM tag IN collection.props WHERE tag.subprop = "value"</pre> <pre>SELECT tag FROM collection c JOIN tag IN c.props WHERE tag.subprop = "value"</pre>
/prop/subprop/?	<p>Index path required to serve queries (with Hash or Range types respectively):</p> <pre>SELECT FROM collection c WHERE c.prop.subprop = "value"</pre> <pre>SELECT FROM collection c WHERE c.prop.subprop > 5</pre>

NOTE

While setting custom index paths, you are required to specify the default indexing rule for the entire document tree denoted by the special path "/*".

The following example configures a specific path with range indexing and a custom precision value of 20 bytes:

```

var collection = new DocumentCollection { Id = "rangeSinglePathCollection" };

collection.IndexingPolicy.IncludedPaths.Add(
    new IncludedPath {
        Path = "/Title/?",
        Indexes = new Collection<Index> {
            new RangeIndex(DataType.String) { Precision = 20 } }
    });

// Default for everything else
collection.IndexingPolicy.IncludedPaths.Add(
    new IncludedPath {
        Path = "/*" ,
        Indexes = new Collection<Index> {
            new HashIndex(DataType.String) { Precision = 3 },
            new RangeIndex(DataType.Number) { Precision = -1 }
        }
    });

collection = await client.CreateDocumentCollectionAsync(UriFactory.CreateDatabaseUri("db"), pathRange);

```

Index data types, kinds and precisions

Now that we've taken a look at how to specify paths, let's look at the options we can use to configure the indexing policy for a path. You can specify one or more indexing definitions for every path:

- Data type: **String**, **Number**, **Point**, **Polygon**, or **LineString** (can contain only one entry per data type per path)
- Index kind: **Hash** (equality queries), **Range** (equality, range or Order By queries), or **Spatial** (spatial queries)
- Precision: 1-8 or -1 (Maximum precision) for numbers, 1-100 (Maximum precision) for string

Index kind

DocumentDB supports Hash and Range index kinds for every path (that can be configured for strings, numbers or both).

- **Hash** supports efficient equality and JOIN queries. For most use cases, hash indexes do not need a higher precision than the default value of 3 bytes. Data Type can be String or Number.
- **Range** supports efficient equality queries, range queries (using $>$, $<$, $>=$, $<=$, $!=$), and Order By queries. Order By queries by default also require maximum index precision (-1). Data Type can be String or Number.

DocumentDB also supports the Spatial index kind for every path, that can be specified for the Point, Polygon, or LineString data types. The value at the specified path must be a valid GeoJSON fragment like

```
{"type": "Point", "coordinates": [0.0, 10.0]}
```

- **Spatial** supports efficient spatial (within and distance) queries. Data Type can be Point, Polygon, or LineString.

NOTE

DocumentDB supports automatic indexing of Points, Polygons, and LineStrings.

Here are the supported index kinds and examples of queries that they can be used to serve:

INDEX KIND	DESCRIPTION/USE CASE
------------	----------------------

INDEX KIND	DESCRIPTION/USE CASE
Hash	<p>Hash over /prop/? (or /) can be used to serve the following queries efficiently:</p> <pre>SELECT FROM collection c WHERE c.prop = "value"</pre> <p>Hash over /props/[]/? (or / or /props/) can be used to serve the following queries efficiently:</p> <pre>SELECT tag FROM collection c JOIN tag IN c.props WHERE tag = 5</pre>
Range	<p>Range over /prop/? (or /) can be used to serve the following queries efficiently:</p> <pre>SELECT FROM collection c WHERE c.prop = "value"</pre> <pre>SELECT FROM collection c WHERE c.prop > 5</pre> <pre>SELECT FROM collection c ORDER BY c.prop</pre>
Spatial	<p>Range over /prop/? (or /) can be used to serve the following queries efficiently:</p> <pre>SELECT FROM collection c</pre> <pre>WHERE ST_DISTANCE(c.prop, {"type": "Point", "coordinates": [0.0, 10.0]}) < 40</pre> <pre>SELECT FROM collection c WHERE ST_WITHIN(c.prop, {"type": "Polygon", ... }) --with indexing on points enabled</pre> <pre>SELECT FROM collection c WHERE ST_WITHIN({"type": "Point", ... }, c.prop) --with indexing on polygons enabled</pre>

By default, an error is returned for queries with range operators such as \geq if there is no range index (of any precision) in order to signal that a scan might be necessary to serve the query. Range queries can be performed without a range index using the `x-ms-documentdb-enable-scan` header in the REST API or the `EnableScanInQuery` request option using the .NET SDK. If there are any other filters in the query that DocumentDB can use the index to filter against, then no error will be returned.

The same rules apply for spatial queries. By default, an error is returned for spatial queries if there is no spatial index, and there are no other filters that can be served from the index. They can be performed as a scan using `x-ms-documentdb-enable-scan/EnableScanInQuery`.

Index precision

Index precision lets you tradeoff between index storage overhead and query performance. For numbers, we recommend using the default precision configuration of -1 ("maximum"). Since numbers are 8 bytes in JSON, this is equivalent to a configuration of 8 bytes. Picking a lower value for precision, such as 1-7, means that values within some ranges map to the same index entry. Therefore you will reduce index storage space, but query execution might have to process more documents and consequently consume more throughput i.e., request units.

Index precision configuration has more practical application with string ranges. Since strings can be any arbitrary length, the choice of the index precision can impact the performance of string range queries, and impact the amount of index storage space required. String range indexes can be configured with 1-100 or -1 ("maximum"). If you would like to perform Order By queries against string properties, then you must specify a precision of -1 for the corresponding paths.

Spatial indexes always use the default index precision for all types (Points, LineStrings, and Polygons) and cannot be overridden.

The following example shows how to increase the precision for range indexes in a collection using the .NET SDK.

Create a collection with a custom index precision

```
var rangeDefault = new DocumentCollection { Id = "rangeCollection" };

// Override the default policy for Strings to range indexing and "max" (-1) precision
rangeDefault.IndexingPolicy = new IndexingPolicy(new RangeIndex(DataType.String) { Precision = -1 });

await client.CreateDocumentCollectionAsync(UriFactory.CreateDatabaseUri("db"), rangeDefault);
```

NOTE

DocumentDB returns an error when a query uses Order By but does not have a range index against the queried path with the maximum precision.

Similarly, paths can be completely excluded from indexing. The next example shows how to exclude an entire section of the documents (a.k.a. a sub-tree) from indexing using the "*" wildcard.

```
var collection = new DocumentCollection { Id = "excludedPathCollection" };
collection.IndexingPolicy.IncludedPaths.Add(new IncludedPath { Path = "/" });
collection.IndexingPolicy.ExcludedPaths.Add(new ExcludedPath { Path = "/nonIndexedContent/*" });

collection = await client.CreateDocumentCollectionAsync(UriFactory.CreateDatabaseUri("db"), excluded);
```

Opting in and opting out of indexing

You can choose whether you want the collection to automatically index all documents. By default, all documents are automatically indexed, but you can choose to turn it off. When indexing is turned off, documents can be accessed only through their self-links or by queries using ID.

With automatic indexing turned off, you can still selectively add only specific documents to the index. Conversely, you can leave automatic indexing on and selectively choose to exclude only specific documents. Indexing on/off configurations are useful when you have only a subset of documents that need to be queried.

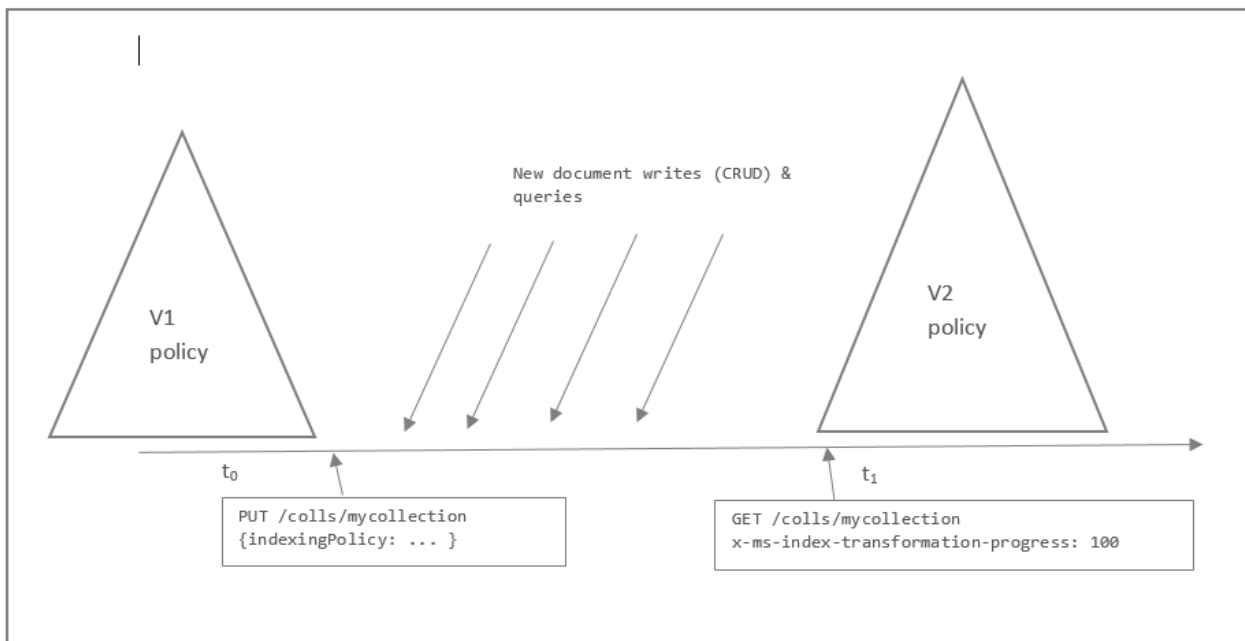
For example, the following sample shows how to include a document explicitly using the [DocumentDB .NET SDK](#) and the [RequestOptions.IndexingDirective](#) property.

```
// If you want to override the default collection behavior to either
// exclude (or include) a Document from indexing,
// use the RequestOptions.IndexingDirective property.
client.CreateDocumentAsync(UriFactory.CreateDocumentCollectionUri("db", "coll"),
    new { id = "AndersenFamily", isRegistered = true },
    new RequestOptions { IndexingDirective = IndexingDirective.Include });
```

Modifying the indexing policy of a collection

DocumentDB allows you to make changes to the indexing policy of a collection on the fly. A change in indexing policy on a DocumentDB collection can lead to a change in the shape of the index including the paths can be indexed, their precision, as well as the consistency model of the index itself. Thus a change in indexing policy, effectively requires a transformation of the old index into a new one.

Online Index Transformations



Index transformations are made online, meaning that the documents indexed per the old policy are efficiently transformed per the new policy **without affecting the write availability or the provisioned throughput** of the collection. The consistency of read and write operations made using the REST API, SDKs or from within stored procedures and triggers is not impacted during index transformation. This means that there is no performance degradation or downtime to your apps when you make an indexing policy change.

However, during the time that index transformation is progress, queries are eventually consistent regardless of the indexing mode configuration (Consistent or Lazy). This also applies to queries from all interfaces – REST API, SDKs, and from within stored procedures and triggers. Just like with Lazy indexing, index transformation is performed asynchronously in the background on the replicas using the spare resources available for a given replica.

Index transformations are also made **in-situ** (in place), i.e. DocumentDB does not maintain two copies of the index and swap the old index out with the new one. This means that no additional disk space is required or consumed in your collections while performing index transformations.

When you change indexing policy, how the changes are applied to move from the old index to the new one depend primarily on the indexing mode configurations more so than the other values like included/excluded paths, index kinds and precisions. If both your old and new policies use consistent indexing, then DocumentDB performs an online index transformation. You cannot apply another indexing policy change with consistent indexing mode while the transformation is in progress.

You can however move to Lazy or None indexing mode while a transformation is in progress.

- When you move to Lazy, the index policy change is made effective immediately and DocumentDB starts recreating the index asynchronously.
- When you move to None, then the index is dropped effective immediately. Moving to None is useful when you want to cancel an in progress transformation and start fresh with a different indexing policy.

If you're using the .NET SDK, you can kick off an indexing policy change using the new **ReplaceDocumentCollectionAsync** method and track the percentage progress of the index transformation using the **IndexTransformationProgress** response property from a **ReadDocumentCollectionAsync** call. Other SDKs and the REST API support equivalent properties and methods for making indexing policy changes.

Here's a code snippet that shows how to modify a collection's indexing policy from Consistent indexing mode to Lazy.

Modify Indexing Policy from Consistent to Lazy

```
// Switch to lazy indexing.
Console.WriteLine("Changing from Default to Lazy IndexingMode.");

collection.IndexingPolicy.IndexingMode = IndexingMode.Lazy;

await client.ReplaceDocumentCollectionAsync(collection);
```

You can check the progress of an index transformation by calling `ReadDocumentCollectionAsync`, for example, as shown below.

Track Progress of Index Transformation

```
long smallWaitTimeMilliseconds = 1000;
long progress = 0;

while (progress < 100)
{
    ResourceResponse<DocumentCollection> collectionReadResponse = await client.ReadDocumentCollectionAsync(
        UriFactory.CreateDocumentCollectionUri("db", "coll"));

    progress = collectionReadResponse.IndexTransformationProgress;

    await Task.Delay(TimeSpan.FromMilliseconds(smallWaitTimeMilliseconds));
}
```

You can drop the index for a collection by moving to the `None` indexing mode. This might be a useful operational tool if you want to cancel an in-progress transformation and start a new one immediately.

Dropping the index for a collection

```
// Switch to lazy indexing.
Console.WriteLine("Dropping index by changing to to the None IndexingMode.");

collection.IndexingPolicy.IndexingMode = IndexingMode.None;

await client.ReplaceDocumentCollectionAsync(collection);
```

When would you make indexing policy changes to your DocumentDB collections? The following are the most common use cases:

- Serve consistent results during normal operation, but fall back to lazy indexing during bulk data imports
- Start using new indexing features on your current DocumentDB collections, e.g., like geospatial querying which require the Spatial index kind, or Order By/string range queries which require the string Range index kind
- Hand select the properties to be indexed and change them over time
- Tune indexing precision to improve query performance or reduce storage consumed

NOTE

To modify indexing policy using `ReplaceDocumentCollectionAsync`, you need version $\geq 1.3.0$ of the .NET SDK

For index transformation to complete successfully, you must ensure that there is sufficient free storage space available on the collection. If the collection reaches its storage quota, then the index transformation will be paused. Index transformation will automatically resume once storage space is available, e.g. if you delete some documents.

Performance tuning

The DocumentDB APIs provide information about performance metrics such as the index storage used, and the

throughput cost (request units) for every operation. This information can be used to compare various indexing policies and for performance tuning.

To check the storage quota and usage of a collection, run a HEAD or GET request against the collection resource, and inspect the `x-ms-request-quota` and the `x-ms-request-usage` headers. In the .NET SDK, the [DocumentSizeQuota](#) and [DocumentSizeUsage](#) properties in [ResourceResponse](#) contain these corresponding values.

```
// Measure the document size usage (which includes the index size) against
// different policies.
ResourceResponse<DocumentCollection> collectionInfo = await
client.ReadDocumentCollectionAsync(UriFactory.CreateDocumentCollectionUri("db", "coll"));
Console.WriteLine("Document size quota: {0}, usage: {1}", collectionInfo.DocumentQuota,
collectionInfo.DocumentUsage);
```

To measure the overhead of indexing on each write operation (create, update, or delete), inspect the `x-ms-request-charge` header (or the equivalent [RequestCharge](#) property in [ResourceResponse](#) in the .NET SDK) to measure the number of request units consumed by these operations.

```
// Measure the performance (request units) of writes.
ResourceResponse<Document> response = await
client.CreateDocumentAsync(UriFactory.CreateDocumentCollectionUri("db", "coll"), myDocument);
Console.WriteLine("Insert of document consumed {0} request units", response.RequestCharge);

// Measure the performance (request units) of queries.
IDocumentQuery<dynamic> queryable = client.CreateDocumentQuery(UriFactory.CreateDocumentCollectionUri("db",
"coll"), queryString).AsDocumentQuery();

double totalRequestCharge = 0;
while (queryable.HasMoreResults)
{
    FeedResponse<dynamic> queryResponse = await queryable.ExecuteNextAsync<dynamic>();
    Console.WriteLine("Query batch consumed {0} request units", queryResponse.RequestCharge);
    totalRequestCharge += queryResponse.RequestCharge;
}

Console.WriteLine("Query consumed {0} request units in total", totalRequestCharge);
```

Changes to the indexing policy specification

A change in the schema for indexing policy was introduced on July 7, 2015 with REST API version 2015-06-03. The corresponding classes in the SDK versions have new implementations to match the schema.

The following changes were implemented in the JSON specification:

- Indexing Policy supports Range indexes for strings
- Each path can have multiple index definitions, one for each data type
- Indexing precision supports 1-8 for numbers, 1-100 for strings, and -1 (maximum precision)
- Paths segments do not require a double quotation to escape each path. For example, you can add a path for `/title/?` instead of `/"title"/?`
- The root path representing "all paths" can be represented as `/*` (in addition to `/`)

If you have code that provisions collections with a custom indexing policy written with version 1.1.0 of the .NET SDK or older, you will need to change your application code to handle these changes in order to move to SDK version 1.2.0. If you do not have code that configures indexing policy, or plan to continue using an older SDK version, no changes are required.

For a practical comparison, here is one example custom indexing policy written using the REST API version 2015-06-03 as well as the previous version 2015-04-08.

Previous Indexing Policy JSON

```
{
  "automatic":true,
  "indexingMode":"Consistent",
  "IncludedPaths":[
    {
      "IndexType":"Hash",
      "Path":"/",
      "NumericPrecision":7,
      "StringPrecision":3
    }
  ],
  "ExcludedPaths":[
    "/\\nonIndexedContent\\/*"
  ]
}
```

Current Indexing Policy JSON

```
{
  "automatic":true,
  "indexingMode":"Consistent",
  "includedPaths":[
    {
      "path":"//*",
      "indexes":[
        {
          "kind":"Hash",
          "dataType":"String",
          "precision":3
        },
        {
          "kind":"Hash",
          "dataType":"Number",
          "precision":7
        }
      ]
    }
  ],
  "ExcludedPaths":[
    {
      "path":"/nonIndexedContent/*"
    }
  ]
}
```

Next Steps

Follow the links below for index policy management samples and to learn more about DocumentDB's query language.

1. [DocumentDB .NET Index Management code samples](#)
2. [DocumentDB REST API Collection Operations](#)
3. [Query with DocumentDB SQL](#)

Securing access to DocumentDB data

11/22/2016 • 6 min to read • [Edit on GitHub](#)

Contributors

Kirat Pandya • mimig • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • James Dunn • Han Wong • arramac • Ryan CrawCour
• v-aljenk • Stephen Baron

This article provides an overview of securing access to data stored in [Microsoft Azure DocumentDB](#).

After reading this overview, you'll be able to answer the following questions:

- What are DocumentDB master keys?
- What are DocumentDB read-only keys?
- What are DocumentDB resource tokens?
- How can I use DocumentDB users and permissions to secure access to DocumentDB data?

DocumentDB access control concepts

DocumentDB provides first class concepts in order to control access to DocumentDB resources. For the purposes of this topic, DocumentDB resources are grouped into two categories:

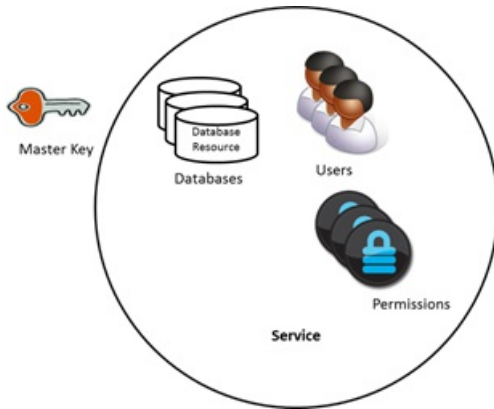
- Administrative resources
 - Account
 - Database
 - User
 - Permission
- Application resources
 - Collection
 - Offer
 - Document
 - Attachment
 - Stored procedure
 - Trigger
 - User-defined function

In the context of these two categories, DocumentDB supports three types of access control personas: account administrator, read-only administrator, and database user. The rights for each access control persona are:

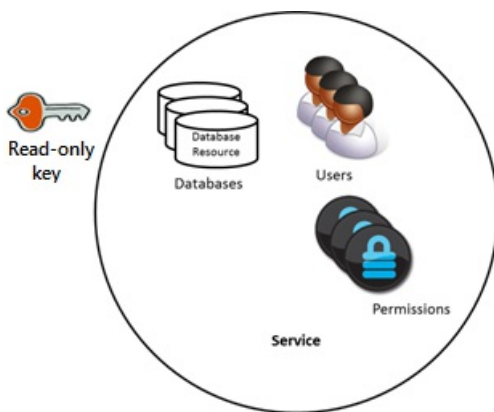
- Account administrator: Full access to all of the resources (administrative and application) within a given DocumentDB account.
- Read-only administrator: Read-only access to all of the resources (administrative and application within a given DocumentDB account.
- Database user: The DocumentDB user resource associated with a specific set of DocumentDB database resources (e.g. collections, documents, scripts). There can be one or more user resources associated with a given database, and each user resource may have one or more permissions associated with it.

With the aforementioned categories and resources in mind, the DocumentDB access control model defines three types of access constructs:

- Master keys: Upon creation of a DocumentDB account, two master keys (primary and secondary) are created. These keys enable full administrative access to all resources within the DocumentDB account.



- Read-only keys: Upon creation of a DocumentDB account, two read-only keys (primary and secondary) are created. These keys enable read-only access to all resources within the DocumentDB account.



- Resource tokens: A resource token is associated with a DocumentDB permission resource and captures the relationship between the user of a database and the permission that user has for a specific DocumentDB application resource (e.g. collection, document).



Working with DocumentDB master and read-only keys

As mentioned earlier, DocumentDB master keys provide full administrative access to all resources within a DocumentDB account, while read-only keys enable read access to all resources within the account. The following code snippet illustrates how to use a DocumentDB account endpoint and master key to instantiate a DocumentClient and create a new database.

```
//Read the DocumentDB endpointUrl and authorization keys from config.
//These values are available from the Azure Classic Portal on the DocumentDB Account Blade under "Keys".
//NB > Keep these values in a safe and secure location. Together they provide Administrative access to your DocDB
account.

private static readonly string endpointUrl = ConfigurationManager.AppSettings["EndPointUrl"];
private static readonly SecureString authorizationKey =
    ToSecureString(ConfigurationManager.AppSettings["AuthorizationKey"]);

client = new DocumentClient(new Uri(endpointUrl), authorizationKey);

// Create Database
Database database = await client.CreateDatabaseAsync(
    new Database
    {
        Id = databaseName
    });
```

Overview of DocumentDB resource tokens

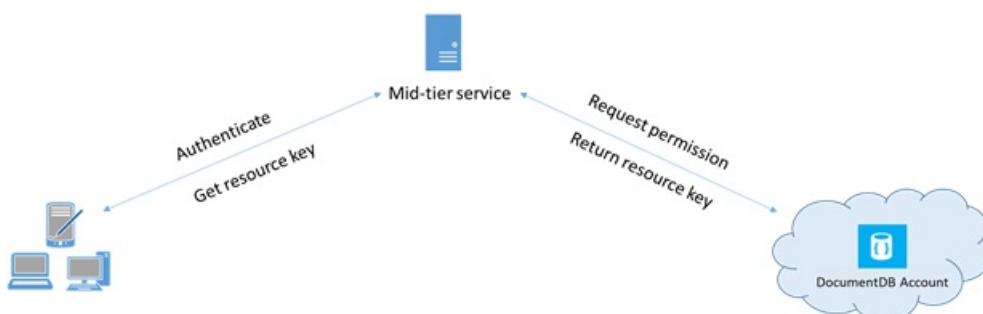
You can use a resource token (by creating DocumentDB users and permissions) when you want to provide access to resources in your DocumentDB account to a client that cannot be trusted with the master key. Your DocumentDB master keys include both a primary and secondary key, each of which grants administrative access to your account and all of the resources in it. Exposing either of your master keys opens your account to the possibility of malicious or negligent use.

Likewise, DocumentDB read-only keys provide read access to all resources - except permission resources, of course - within a DocumentDB account and cannot be used to provide more granular access to specific DocumentDB resources.

DocumentDB resource tokens provide a safe alternative that allows clients to read, write, and delete resources in your DocumentDB account according to the permissions you've granted, and without need for either a master or read only key.

Here is a typical design pattern whereby resource tokens may be requested, generated and delivered to clients:

1. A mid-tier service is set up to serve a mobile application to share user photos.
2. The mid-tier service possesses the master key of the DocumentDB account.
3. The photo app is installed on end user mobile devices.
4. On login, the photo app establishes the identity of the user with the mid-tier service. This mechanism of identity establishment is purely up to the application.
5. Once the identity is established, the mid-tier service requests permissions based on the identity.
6. The mid-tier service sends a resource token back to the phone app.
7. The phone app can continue to use the resource token to directly access DocumentDB resources with the permissions defined by the resource token and for the interval allowed by the resource token.
8. When the resource token expires, subsequent requests will receive a 401 unauthorized exception. At this point, the phone app re-establishes the identity and requests a new resource token.



Working with DocumentDB users and permissions

A DocumentDB user resource is associated with a DocumentDB database. Each database may contain zero or more DocumentDB users. The following code snippet shows how to create a DocumentDB user resource.

```
//Create a user.
User docUser = new User
{
    Id = "mobileuser"
};

docUser = await client.CreateUserAsync(UriFactory.CreateDatabaseUri("db"), docUser);
```

NOTE

Each DocumentDB user has a `PermissionsLink` property which can be used to retrieve the list of permissions associated with the user.

A DocumentDB permission resource is associated with a DocumentDB user. Each user may contain zero or more DocumentDB permissions. A permission resource provides access to a security token that the user needs when trying to access a specific application resource. There are two available access levels which may be provided by a permission resource:

- All: The user has full permission on the resource
- Read: The user can only read the contents of the resource but cannot perform write, update, or delete operations on the resource.

NOTE

In order to run DocumentDB stored procedures the user must have the All permission on the collection in which the stored procedure will be run.

The following code snippet shows how to create a permission resource, read the resource token of the permission resource and associate the permissions with the user created above.

```
// Create a permission.
Permission docPermission = new Permission
{
    PermissionMode = PermissionMode.Read,
    ResourceLink = documentCollection.SelfLink,
    Id = "readperm"
};

docPermission = await client.CreatePermissionAsync(UriFactory.CreateUserUri("db", "user"), docPermission);
Console.WriteLine(docPermission.Id + " has token of: " + docPermission.Token);
```

If you have specified a partition key for your collection, then the permission for collection, document and attachment resources must also include the `ResourcePartitionKey` in addition to the `ResourceLink`.

In order to easily obtain all permission resources associated with a particular user, DocumentDB makes available a permission feed for each user object. The following code snippet shows how to retrieve the permission associated with the user created above, construct a permission list, and instantiate a new `DocumentClient` on behalf of the user.

```
//Read a permission feed.
FeedResponse<Permission> permFeed = await client.ReadPermissionFeedAsync(
    UriFactory.CreateUserUri("db", "myUser"));

List<Permission> permList = new List<Permission>();

foreach (Permission perm in permFeed)
{
    permList.Add(perm);
}

DocumentClient userClient = new DocumentClient(new Uri(endpointUrl), permList);
```

TIP

Resource tokens have a default valid timespan of 1 hour. Token lifetime, however, may be explicitly specified, up to a maximum of 5 hours.

Next steps

- To learn more about DocumentDB, click [here](#).
- To learn about managing master and read-only keys, click [here](#).
- To learn how to construct DocumentDB authorization tokens, click [here](#)

Automatic online backup and restore with DocumentDB

11/15/2016 • 3 min to read • [Edit on GitHub](#)

Contributors

Rahul Prasad • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • mimig

Azure DocumentDB automatically takes backups of all your data at regular intervals. The automatic backups are taken without affecting the performance or availability of your NoSQL database operations. All your backups are stored separately in another storage service, and those backups are globally replicated for resiliency against regional disasters. The automatic backups are intended for scenarios when you accidentally delete your DocumentDB collection and later require data recovery or a disaster recovery solution.

This article starts with a quick recap of the data redundancy and availability in DocumentDB, and then discusses backups.

High availability with DocumentDB - a recap

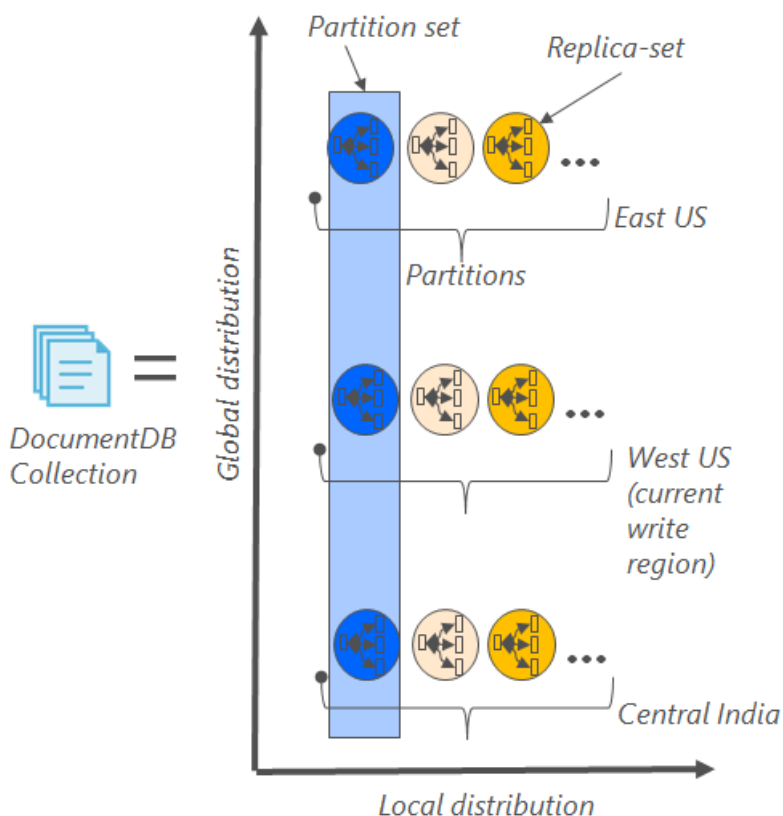
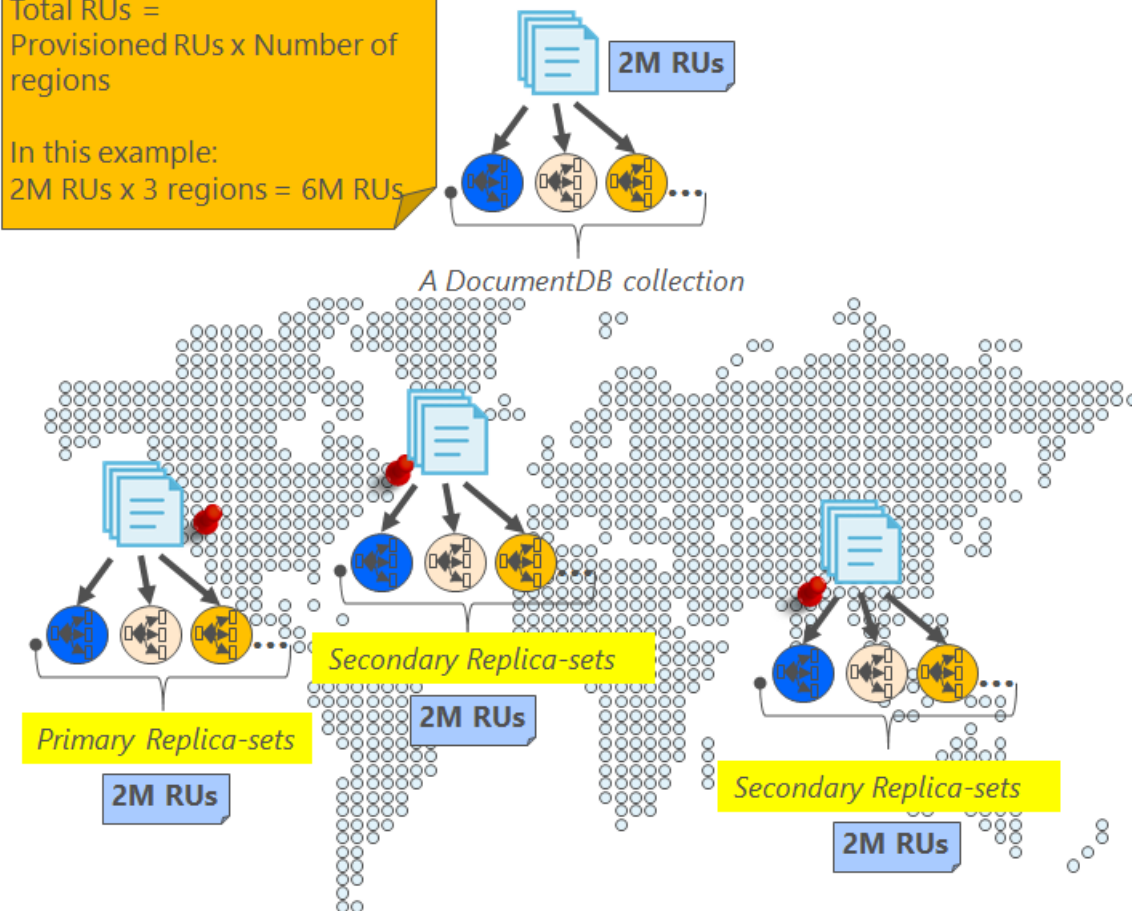
DocumentDB is designed to be [globally distributed](#) – it allows you to scale throughput across multiple Azure regions along with policy driven failover and transparent multi-homing APIs. As a database system offering [99.99% availability SLAs](#), all the writes in DocumentDB are durably committed to local disks by a quorum of replicas within a local data center before acknowledging to the client. Note that the high availability of DocumentDB relies on local storage and does not depend on any external storage technologies. Additionally, if your database account is associated with more than one Azure region, your writes are replicated across other regions as well. To scale your throughput and access data at low latencies, you can have as many read regions associated with your database account as you like. In each read region, the (replicated) data is durably persisted across a replica set.

As illustrated in the following diagram, a single DocumentDB collection is [horizontally partitioned](#). A “partition” is denoted by a circle in the following diagram, and each partition is made highly available via a replica set. This is the local distribution within a single Azure region (denoted by the X axis). Further, each partition (with its corresponding replica set) is then globally distributed across multiple regions associated with your database account (for example, in this illustration the three regions – East US, West US and Central India). The “partition set” is a globally distributed entity comprising of multiple copies of your data in each region (denoted by the Y axis). You can assign priority to the regions associated with your database account and DocumentDB will transparently failover to the next region in case of disaster. You can also manually simulate failover to test the end-to-end availability of your application.

The following image illustrates the high degree of redundancy with DocumentDB.

Total RUs =
Provisioned RUs x Number of
regions

In this example:
2M RUs x 3 regions = 6M RUs



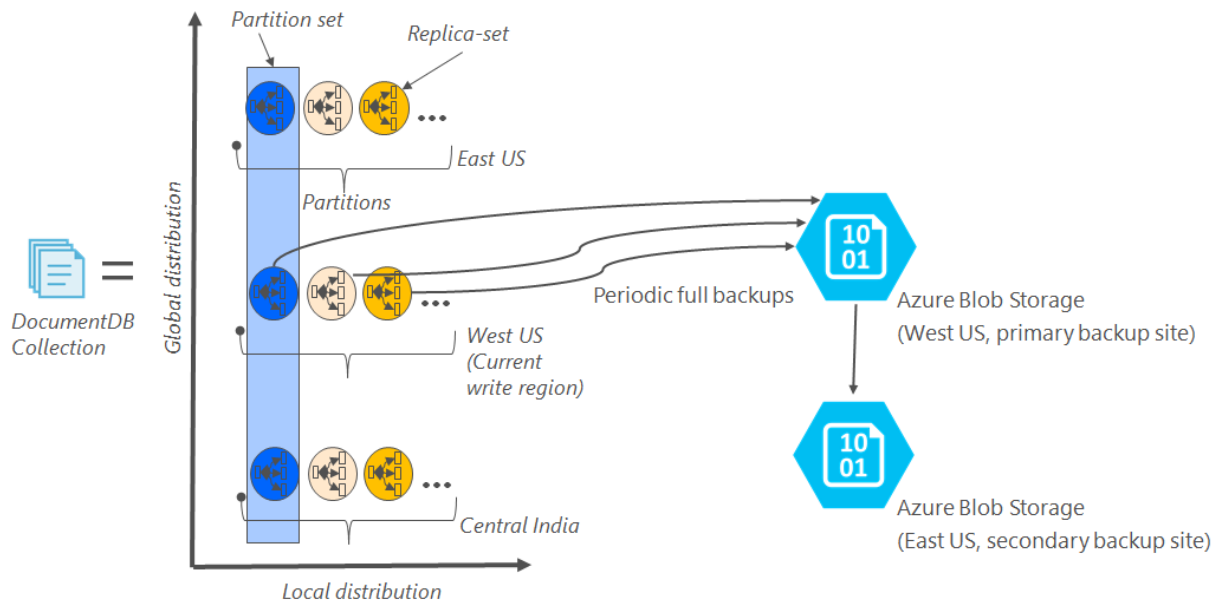
Full, automatic, online backups

Oops, I deleted my collection or database! With DocumentDB, not only your data, but the backups of your data are also made highly redundant and resilient to regional disasters. These automated backups are currently taken approximately every four hours.

The backups are taken without affecting the performance or availability of your database operations. DocumentDB takes the backup in the background without consuming your provisioned RUs or affecting the performance and without affecting the availability of your NoSQL database.

Unlike your data that is stored inside DocumentDB, the automatic backups are stored in Azure Blob Storage service. To guarantee the low latency/efficient upload, the snapshot of your backup is uploaded to an instance of Azure Blob storage in the same region as the current write region of your DocumentDB database account. For resiliency against regional disaster, each snapshot of your backup data in Azure Blob Storage is again replicated via geo-redundant storage (GRS) to another region. The following diagram shows that the entire DocumentDB collection (with all three primary partitions in West US, in this example) is backed up in a remote Azure Blob Storage account in West US and then GRS replicated to East US.

The following image illustrates periodic full backups of all DocumentDB entities in GRS Azure Storage.



Retention period for a given snapshot

As described above, we periodically take snapshots of your data and per our compliance regulations, we retain the latest snapshot up to 90 days before it eventually gets purged. If a collection or account is deleted, DocumentDB stores the last backup for 90 days.

Restore database from the online backup

In case you accidentally delete your data, you can [file a support ticket](#) or [call Azure support](#) to restore the data from the last automatic backup. For a specific snapshot of your backup to be restored, DocumentDB requires that the data was at least available with us for the duration of the backup cycle for that snapshot.

Next steps

To replicate your NoSQL database in multiple data centers, see [distribute your data globally with DocumentDB](#).

To file contact Azure Support, [file a ticket from the Azure portal](#).

Performance levels and pricing tiers in DocumentDB

11/15/2016 • 8 min to read • [Edit on GitHub](#)

Contributors

[mimig](#) • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [PRmerger](#) • [Jason Card](#) • [arramac](#) • [Carolyn Gronlund](#) • [John Macintyre](#) • [v-aljenk](#) • [Dene Hager](#)

This article provides an overview of performance levels in [Microsoft Azure DocumentDB](#).

After reading this article, you'll be able to answer the following questions:

- What is a performance level?
- How is throughput reserved for a database account?
- How do I work with performance levels?
- How am I billed for performance levels?

Introduction to performance levels

Each DocumentDB collection created in a Standard DocumentDB account is provisioned with an associated performance level. Each collection in a database can have a different performance level enabling you to designate more throughput for frequently accessed collections and less throughput for infrequently accessed collections.

DocumentDB supports both **user-defined** performance levels and **pre-defined** performance levels, as shown in the following table. User-defined performance enables you to reserved throughput in units of 100 RU/s and have unlimited storage, whereas the three pre-defined performance levels have specified throughput options, and a 10GB storage quota. The following table compares **user-defined** performance to **pre-defined** performance.

PERFORMANCE TYPE	DETAILS	THROUGHPUT	STORAGE	VERSION	APIS
User-defined performance	User sets throughput in units of 100 RU/s	Unlimited. 400 - 250,000 request units/s by default (higher by request)	Unlimited. 250 GB by default (higher by request)	V2	API 2015-12-16 and newer
Pre-defined performance	10 GB reserved storage. S1 = 250 RU/s S2 = 1000 RU/s S3 = 2500 RU/s	2500 RU/s	10 GB	V1	Any

Throughput is reserved per collection, and is available for use by that collection exclusively. Throughput is measured in [request units \(RUs\)](#), which identify the amount of resources required to perform various DocumentDB database operations.

NOTE

The performance level of a collection can be adjusted through the [SDKs](#) or the [Azure portal](#). Performance level changes are expected to complete within 3 minutes.

Setting performance levels for collections

Once a collection is created, the full allocation of RUs based on the designated performance level are reserved for the collection.

Note that with both user-defined and pre-defined performance levels, DocumentDB operates based on reservation of throughput. By creating a collection, an application has reserved and is billed for reserved throughput regardless of how much of that throughput is actively used. With user-defined performance levels, storage is metered based on consumption, but with pre-defined performance levels, 10 GB of storage is reserved at the time of collection creation.

After collections are created, you can modify the performance level and/or throughput by using the [SDKs](#) or the [Azure portal](#).

IMPORTANT

DocumentDB Standard collections are billed at an hourly rate and each collection you create will be billed for a minimum one hour of usage.

If you adjust the performance level of a collection within an hour, you will be billed for the highest performance level set during the hour. For example, if you increase your performance level for a collection at 8:53am you will be charged for the new level starting at 8:00am. Likewise, if you decrease your performance level at 8:53am, the new rate will be applied at 9:00am.

Request units are reserved for each collection based on the performance level set. Request unit consumption is evaluated as a per second rate. Applications that exceed the provisioned request unit rate (or performance level) on a collection will be throttled until the rate drops below the reserved level for that collection. If your application requires a higher level of throughput, you can increase the performance level for each collection.

NOTE

When your application exceeds performance levels for one or multiple collections, requests will be throttled on a per collection basis. This means that some application requests may succeed while others may be throttled. It is recommended to add a small number of retries when throttled in order to handle spikes in request traffic.

Working with performance levels

DocumentDB collections enable you to group your data based on both the query patterns and performance needs of your application. With DocumentDB's automatic indexing and query support, it is quite common to collocate heterogeneous documents within the same collection. The key considerations in deciding whether separate collections should be used include:

- Queries – A collection is the scope for query execution. If you need to query across a set of documents, the most efficient read patterns come from collocating documents in a single collection.
- Transactions – All transactions are scoped to within a single collection. If you have documents that must be updated within a single stored procedure or trigger, they must be stored within the same collection. More specifically, a partition key within a collection is the transaction boundary. Please see [Partitioning in DocumentDB](#) for more details.

- Performance isolation – A collection has an associated performance level. This ensures that each collection has a predictable performance through reserved RUs. Data can be allocated to different collections, with different performance levels, based on access frequency.

IMPORTANT

It is important to understand you will be billed at full standard rates based on the number of collections created by your application.

It is recommended that your application makes use of a small number of collections unless you have large storage or throughput requirements. Ensure that you have well understood application patterns for the creation of new collections. You may choose to reserve collection creation as a management action handled outside your application. Similarly, adjusting the performance level for a collection will change the hourly rate at which the collection is billed. You should monitor collection performance levels if your application adjusts these dynamically.

Change from S1, S2, S3 to user-defined performance

Follow these steps to change from using pre-defined throughput levels to user-defined throughput levels in the Azure portal. By using user-defined throughput levels, you can tailor your throughput to your needs. And if you're still using an S1 account, you can increase your default throughput from 250 RU/s to 400 RU/s with just a few clicks. Note that once you move a collection from S1, S2 or S3 to Standard (user-defined), you cannot move back to S1, S2, or S3, you can however modify the throughput of a Standard collection at any time.

For more information about the pricing changes related to user-defined and pre-defined throughput, see the blog post [DocumentDB: Everything you need to know about using the new pricing options](#).

1. In the [Azure portal](#), click **NoSQL (DocumentDB)**, then select the DocumentDB account to modify.

If **NoSQL (DocumentDB)** is not on the Jumpbar, click >, scroll to **Databases**, select **NoSQL (DocumentDB)**, and then select the DocumentDB account.

2. On the resource menu, under **Collections**, click **Scale**, select the collection to modify from the drop down list, and then click **Pricing Tier**. Accounts using pre-defined throughput have a pricing tier of S1, S2, or S3. In the **Choose your pricing tier** blade, click **Standard** to change to user-defined throughput, and then click **Select** to save your change.

The screenshot shows the Azure portal interface for scaling a Cosmos DB collection. The left sidebar has the 'Scale' option highlighted. The main area shows the current configuration: Pricing Tier is S1, Throughput (RU/s) is 250, and the Estimated Monthly Bill is 25 USD. A dialog box titled 'Choose your pricing tier' is open on the right, displaying three pricing tiers: STANDARD (24.00 USD/month), S1 (25.00 USD/month), and S3 (100.00 USD/month). The STANDARD tier is highlighted with a red box.

- Back in the **Scale** blade, the **Pricing Tier** is changed to **Standard** and the **Throughput (RU/s)** box is displayed with a default value of 400. Set the throughput between 400 and 10,000 [Request units](#)/second (RU/s). The **Estimated Monthly Bill** at the bottom of the page updates automatically to provide an estimate of the monthly cost. Click **Save** to save your changes.

If you determine that you need more throughput (greater than 10,000 RU/s) or more storage (greater than 10GB) you can create a partitioned collection. To create a partitioned collection, see [Create a collection](#).

NOTE

Changing performance levels of a collection may take up to 2 minutes.

Changing performance levels using the .NET SDK

Another option for changing your collections' performance levels is through our SDKs. This section only covers changing a collection's performance level using our [.NET SDK](#), but the process is similar for our other [SDKs](#). If you are new to our .NET SDK, please visit our [getting started tutorial](#).

Here is a code snippet for changing the offer throughput to 50,000 request units per second:

```
//Fetch the resource to be updated
Offer offer = client.CreateOfferQuery()
    .Where(r => r.ResourceLink == collection.SelfLink)
    .AsEnumerable()
    .SingleOrDefault();

// Set the throughput to 5000 request units per second
offer = new OfferV2(offer, 5000);

//Now persist these changes to the database by replacing the original resource
await client.ReplaceOfferAsync(offer);

// Set the throughput to S2
offer = new Offer(offer);
offer.OfferType = "S2";

//Now persist these changes to the database by replacing the original resource
await client.ReplaceOfferAsync(offer);
```

NOTE

Collections provisioned with under 10,000 request units per second can be migrated between offers with user-defined throughput and pre-defined throughput (S1, S2, S3) at any time. Collections which are provisioned with above 10,000 request units per second cannot be converted to pre-defined throughput levels.

Visit [MSDN](#) to view additional examples and learn more about our offer methods:

- [ReadOfferAsync](#)
- [ReadOffersFeedAsync](#)
- [ReplaceOfferAsync](#)
- [CreateOfferQuery](#)

Changing the throughput of a collection

If you are already using user-defined performance, you can change the throughput of your collection by doing the following. If you need to change from an S1, S2 or S3 performance level (pre-defined performance) to user-defined performance, see [Change from S1, S2, S3 to user-defined performance](#).

1. In the [Azure portal](#), click **NoSQL (DocumentDB)**, then select the DocumentDB account to modify.
2. On the resource menu, under **Collections**, click **Scale**, select the collection to modify from the drop down list.
3. In the **Throughput (RU/s)** box, type the new throughput level.

The **Estimated Monthly Bill** at the bottom of the page updates automatically to provide an estimate of the monthly cost. Click **Save** to save your changes.

If you're not sure how much to increase your throughput, see [Estimating throughput needs](#) and the [Request unit calculator](#).

Troubleshooting

If you do not see the option to change between S1, S2, or S3 performance levels on the **Choose your pricing tier** blade, click **View all** to display the Standard, S1, S2, and S3 performance levels. If you are using the Standard pricing tier, you cannot change between S1, S2, and S3.

Choose your pricing tier

DocumentDB supports a simple and flexible pricing model. Collections are billed based on the volume of data stored and the throughput provisioned.
[Learn more](#)

★ Recommended

View all

STANDARD	S1	S2
6.00 USD per 100 RU/s	250 RUs	1K RUs
0.25 USD per GB used		
Single Partition Up to 10k RU/s & 10GB	Single Partition	Single Partition
Partitioned Unlimited RU/s and storage	10 GB Storage	10 GB Storage
99.99% Availability SLA	99.99% Availability SLA	99.99% Availability SLA
24.00 STARTING COST (USD)/MONTH	25.00 USD/MONTH (ESTIMATED)	50.00 USD/MONTH (ESTIMATED)

S3

2.5K RUs

Single Partition

10 GB Storage

99.99% Availability SLA

100.00
USD/MONTH (ESTIMATED)

Select

Once you change a collection from S1, S2, or S3 to Standard, you cannot move back to S1, S2, or S3.

Next steps

To learn more about pricing and managing data with Azure DocumentDB, explore these resources:

- [DocumentDB pricing](#)
- [Managing DocumentDB capacity](#)
- [Modeling data in DocumentDB](#)
- [Partitioning data in DocumentDB](#)
- [Request units](#)

To learn more about DocumentDB, see the Azure DocumentDB [documentation](#).

To get started with scale and performance testing with DocumentDB, see [Performance and Scale Testing with Azure DocumentDB](#).

Default quotas for DocumentDB

11/22/2016 • 1 min to read • [Edit on GitHub](#)

Contributors

[mimig](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [arramac](#) • [Andrew Hoh](#) • [Han Wong](#) • [Andy Pasic](#)

The following table describes the default quotas for Azure DocumentDB database resources.

ENTITY	DEFAULT QUOTA (STANDARD OFFER)
Document storage per collection	250 GB*
Throughput per collection, measured in Request Units per second per collection	250,000 RU/s*

Quotas listed with an asterisk (*) [can be adjusted by contacting Azure support](#). Quota increases may take up to 24 hours to complete after receiving the required information.

Request increased DocumentDB account quotas

11/15/2016 • 1 min to read • [Edit on GitHub](#)

Contributors

[Andrew Hoh](#) • [mimig](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [arramac](#) • [v-aljenk](#) • [Stephen Baron](#)

[Microsoft Azure DocumentDB](#) has a set of default quotas that can be adjusted by contacting Azure support. This article shows how to request a quota increase.

After reading this article, you'll be able to answer the following questions:

- Which DocumentDB database quotas can be adjusted by contacting Azure support?
- How can I request a DocumentDB account quota adjustment?

DocumentDB account quotas

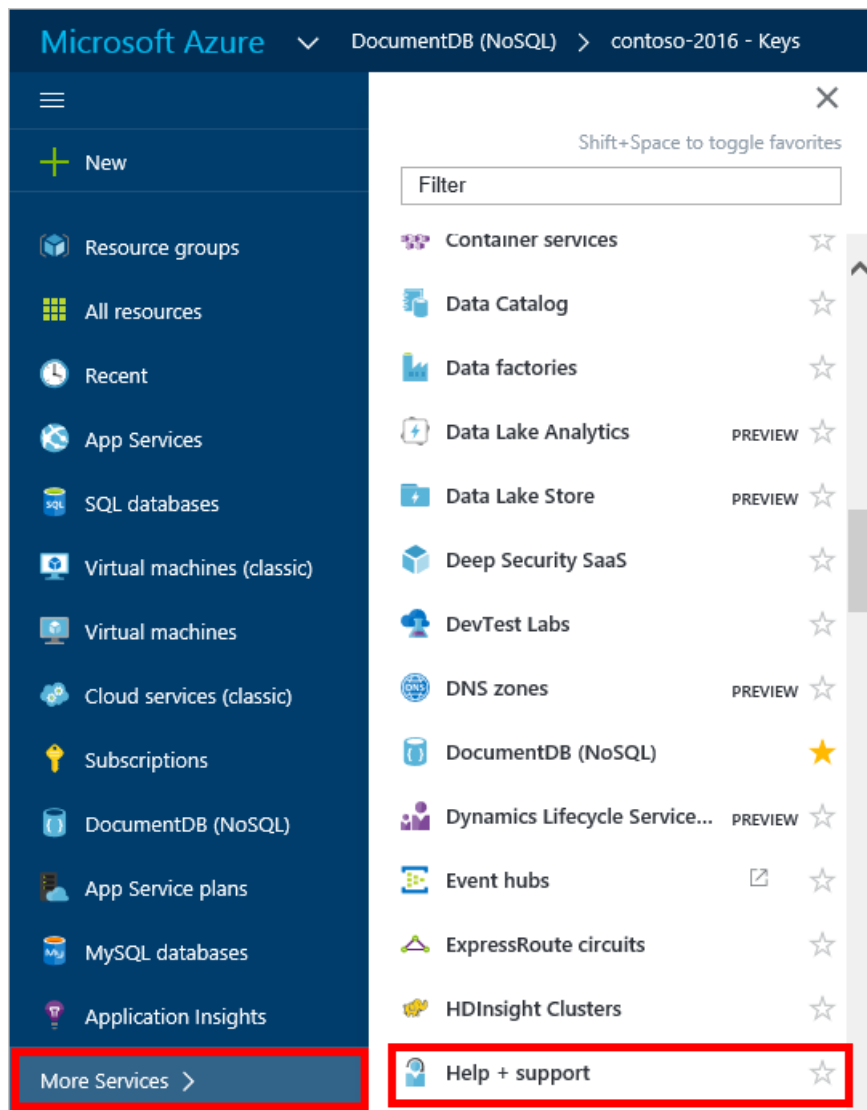
The following table describes the DocumentDB quotas. The quotas that have an asterisk (*) can be adjusted by contacting Azure support:

ENTITY	DEFAULT QUOTA (STANDARD OFFER)
Document storage per collection	250 GB*
Throughput per collection, measured in Request Units per second per collection	250,000 RU/s*

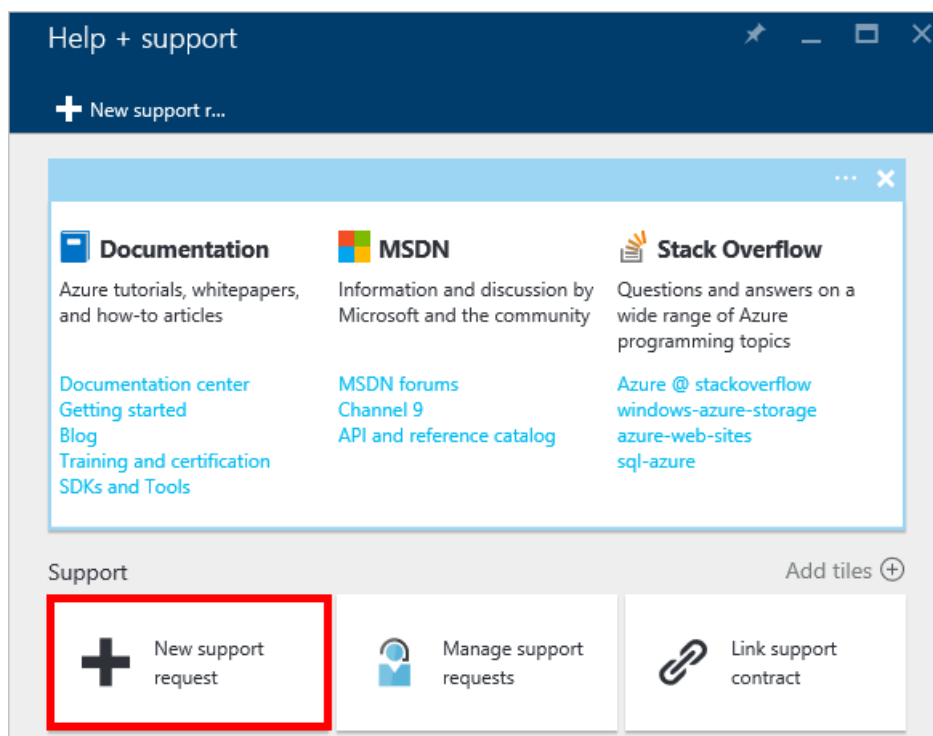
Request a quota adjustment

The following steps show how to request a quota adjustment.

1. In the [Azure portal](#), click **More Services**, and then click **Help + support**.



2. In the **Help + support** blade, click **New support request**.



3. In the **New support request** blade, click **Basics**. Next, set **Issue type** to **Quota**, **Subscription** to your subscription that hosts your DocumentDB account, **Quota type** to **DocumentDB**, and **Support plan** to **Quota SUPPORT - Included**. Then, click **Next**.

New support req... HELP + SUPPORT

Basics NEW SUPPORT REQUEST

1 Basics >

2 Problem >

3 Contact information >

* Issue type
Quota

* Subscription
Visual Studio Ultimate with MSDN (2c60...)

* Quota type
DocumentDB

* Support plan ⓘ
Quota support - Included

Next

4. In the **Problem** blade, choose a severity and include information about your quota increase in **Details**. Click **Next**.

New support req... HELP + SUPPORT

Problem NEW SUPPORT REQUEST

1 Basics ✓

2 Problem >

3 Contact information >

* Severity ⓘ
C - Minimal impact

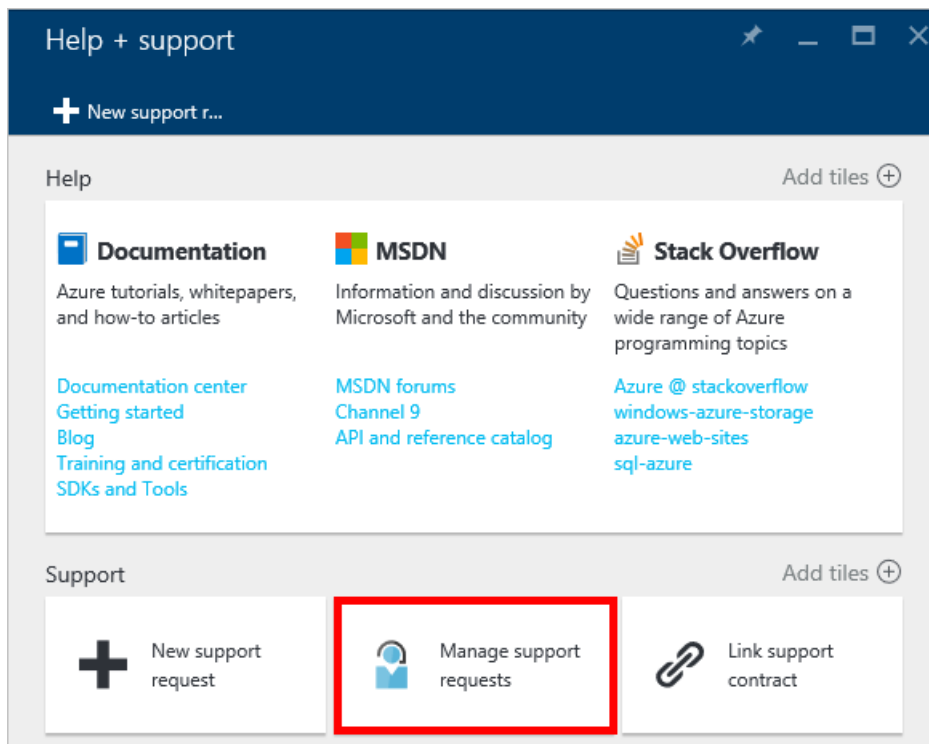
* Details
I would like to increase the following DocumentDB quotas on DocumentDB account "contosodocdb":
Account: contosodocdb
Document storage per collection: 1 TB

File upload ⓘ
Select a file

Next

5. Finally, fill in your contact information in the **Contact information** blade and click **Create**.

Once the support ticket has been created, you should receive the support request number via email. You can also view the support request by clicking **Manage support requests** in the **Help + support** blade.



Next steps

- To learn more about DocumentDB, click [here](#).

Request Units in DocumentDB

11/22/2016 • 8 min to read • [Edit on GitHub](#)

Contributors

Syam Nair • mimig • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Stephen Baron • katiecumming • arramac

Now available: DocumentDB [request unit calculator](#). Learn more in [Estimating your throughput needs](#).

Azure DocumentDB

Estimate Request Units and Data Storage

DocumentDB is offered in units of solid-state drive (SSD) backed storage and throughput. Request units measure DocumentDB throughput per second, and request unit consumption varies by operation and JSON document. Use this calculator to determine the number of request units per second (RU/s) and the amount of data storage needed by your application. Read the [Request Units in DocumentDB](#) article for more information.

Add one or more JSON documents that are each representative of one type of document used by your application.

Sample Document 1

Sample JSON document: [Upload Document](#)

Number of documents:

Create / second:

Read / second:

Update / second:

Delete / second:

[+ Add an additional sample document](#)

[Calculate](#)

Estimated Total

Total RUs for create Q/sec

Total RUs for read Q/sec

Total RUs for update Q/sec

Total RUs for delete Q/sec

0 RU/sec

Total Data Storage **0**

[Go to Azure.com for Pricing](#)

Introduction

This article provides an overview of request units in [Microsoft Azure DocumentDB](#).

After reading this article, you'll be able to answer the following questions:

- What are request units and request charges?
- How do I specify request unit capacity for a collection?
- How do I estimate my application's request unit needs?
- What happens if I exceed request unit capacity for a collection?

Request units and request charges

DocumentDB delivers fast, predictable performance by *reserving* resources to satisfy your application's throughput needs. Because application load and access patterns change over time, DocumentDB allows you to easily increase or decrease the amount of reserved throughput available to your application.

With DocumentDB, reserved throughput is specified in terms of request units processing per second. You can think of request units as throughput currency, whereby you *reserve* an amount of guaranteed request units available to your application on per second basis. Each operation in DocumentDB - writing a document, performing a query, updating a document - consumes CPU, memory, and IOPS. That is, each operation incurs a *request charge*, which is expressed in *request units*. Understanding the factors which impact request unit charges, along with your application's throughput requirements, enables you to run your application as cost effectively as possible.

We recommend getting started by watching the following video, where Aravind Ramachandran explains request units and predictable performance with DocumentDB.

Specifying request unit capacity

When creating a DocumentDB collection, you specify the number of request units per second (RUs) you want reserved for the collection. Once the collection is created, the full allocation of RUs specified is reserved for the collection's use. Each collection is guaranteed to have dedicated and isolated throughput characteristics.

It is important to note that DocumentDB operates on a reservation model; that is, you are billed for the amount of throughput *reserved* for the collection, regardless of how much of that throughput is actively *used*. Keep in mind, however, that as your application's load, data, and usage patterns change you can easily scale up and down the amount of reserved RUs through DocumentDB SDKs or using the [Azure Portal](#). For more information on to scale throughput up and down, see [DocumentDB performance levels](#).

Request unit considerations

When estimating the number of request units to reserve for your DocumentDB collection, it is important to take the following variables into consideration:

- **Document size.** As document sizes increase the units consumed to read or write the data will also increase.
- **Document property count.** Assuming default indexing of all properties, the units consumed to write a document will increase as the property count increases.
- **Data consistency.** When using data consistency levels of Strong or Bounded Staleness, additional units will be consumed to read documents.
- **Indexed properties.** An index policy on each collection determines which properties are indexed by default. You can reduce your request unit consumption by limiting the number of indexed properties or by enabling lazy indexing.
- **Document indexing.** By default each document is automatically indexed, you will consume fewer request units if you choose not to index some of your documents.
- **Query patterns.** The complexity of a query impacts how many Request Units are consumed for an operation. The number of predicates, nature of the predicates, projections, number of UDFs, and the size of the source data set all influence the cost of query operations.
- **Script usage.** As with queries, stored procedures and triggers consume request units based on the complexity of the operations being performed. As you develop your application, inspect the request charge header to better understand how each operation is consuming request unit capacity.

Estimating throughput needs

A request unit is a normalized measure of request processing cost. A single request unit represents the processing capacity required to read (via self link or id) a single 1KB JSON document consisting of 10 unique property values (excluding system properties). A request to create (insert), replace or delete the same document will consume more processing from the service and thereby more request units.

NOTE

The baseline of 1 request unit for a 1KB document corresponds to a simple GET by self link or id of the document.

Use the request unit calculator

To help customers fine tune their throughput estimations, there is a web based [request unit calculator](#) to help estimate the request unit requirements for typical operations, including:

- Document creates (writes)
- Document reads
- Document deletes
- Document updates

The tool also includes support for estimating data storage needs based on the sample documents you provide.

Using the tool is simple:

1. Upload one or more representative JSON documents.

The screenshot shows the 'Estimate Request Units and Data Storage' tool for Azure DocumentDB. It features a sidebar for 'Sample Document 1' with input fields for document count and operations per second. The main area displays an 'Estimated Total' of 0 RU/sec and 0 Total Data Storage. A red box highlights the 'Upload Document' button in the sidebar.

Operation	Estimated Total
Total RUs for create	0/sec
Total RUs for read	0/sec
Total RUs for update	0/sec
Total RUs for delete	0/sec
Total RU/sec	0 RU/sec
Total Data Storage	0

2. To estimate data storage requirements, enter the total number of documents you expect to store.
3. Enter the number of document create, read, update, and delete operations you require (on a per-second basis). To estimate the request unit charges of document update operations, upload a copy of the sample document from step 1 above that includes typical field updates. For example, if document updates typically modify two properties named lastLogin and userVisits, then simply copy the sample document, update the values for those two properties, and upload the copied document.

Azure DocumentDB

Add one or more JSON documents that are each representative of one type of document used by your application.

Sample Document 1

Sample JSON document:

[userDocument.json](#)
Remove

Number of documents:

1250000

Create / second:

20

Read / second:

200

Update / second:

5

[userDocument - Replace.json](#)
Remove

Delete / second:

5

Sample Document 2

Sample JSON document:

[postDocument.json](#)
Remove

Number of documents:

12500000

Create / second:

50

Read / second:

500

Update / second:

0

Delete / second:

5

+ Add an additional sample document

Calculate

Estimated Total

Total RUs for create

0/sec

Total RUs for read

0/sec

Total RUs for update

0/sec

Total RUs for delete

0/sec

0 RU/sec

Total Data Storage

0

Go to Azure.com for Pricing

4. Click calculate and examine the results.

Azure DocumentDB

Add one or more JSON documents that are each representative of one type of document used by your application.

Sample Document 1

Sample JSON document:

[userDocument.json](#)
Remove

Number of documents:

1250000

Create / second:

20

Read / second:

200

Update / second:

5

[userDocument - Replace.json](#)
Remove

Delete / second:

5

Sample Document 2

Sample JSON document:

[postDocument.json](#)
Remove

Number of documents:

12500000

Create / second:

50

Read / second:

500

Update / second:

0

Delete / second:

5

+ Add an additional sample document

Calculate

Estimated Total

Total RUs for create

1597/sec

For sample document 1

11.24 RUs x 20 = 225

For sample document 2

27.43 RUs x 50 = 1372

Total RUs for read

725/sec

For sample document 1

1.00 RUs x 200 = 200

For sample document 2

1.05 RUs x 500 = 525

Total RUs for update

60/sec

For sample document 1

11.81 RUs x 5 = 60

Total RUs for delete

198/sec

For sample document 1

12.00 RUs x 5 = 60

For sample document 2

27.43 RUs x 5 = 138

2580 RUs/sec

Total Data Storage

24.34 GB

For sample document 1

0.63 KB x 1250000 = 0.54 GB

For sample document 2

2.10 KB x 12500000 = 23.80 GB

Go to Azure.com for Pricing

NOTE

If you have document types which will differ dramatically in terms of size and the number of indexed properties, then upload a sample of each *type* of typical document to the tool and then calculate the results.

Use the DocumentDB request charge response header

Every response from the DocumentDB service includes a custom header (x-ms-request-charge) that contains the request units consumed for the request. This header is also accessible through the DocumentDB SDKs. In the .NET SDK, RequestCharge is a property of the ResourceResponse object. For queries, the DocumentDB Query Explorer in the Azure portal provides request charge information for executed queries.

Query Explorer

Load File

Settings

Run query

Databases
nutrition

Collections
nutritiondataset

```
SELECT TOP 10 food.id,
    food.description,
    food.tags,
    food.foodGroup
FROM food
WHERE food.foodGroup = "Snacks"
```

Information

REQUEST CHARGE	9.99 RUS
ROUND TRIPS	1
SHOWING RESULTS	1-10

Results

Previous page

Next page

```
[
  {
    "id": "08510",
    "description": "Milk and cereal bar",
    "tags": [
      {
        "name": "milk and cereal bar"
      }
    ],
    "foodGroup": "Snacks"
  },
  {
    "id": "08546",
    "description": "Rice and Wheat cereal bar",
    "tags": [
      {
        "name": "rice and wheat cereal bar"
      }
    ],
    "foodGroup": "Snacks"
  },
  {
    "id": "19034",
    "description": "Snacks, popcorn, air-popped",
    "tags": [
      {
        "name": "snacks"
      },
      {
        "name": "popcorn"
      },
      {
        "name": "air-popped"
      }
    ],
    "foodGroup": "Snacks"
  },
]
```

With this in mind, one method for estimating the amount of reserved throughput required by your application is to record the request unit charge associated with running typical operations against a representative document used by your application and then estimating the number of operations you anticipate performing each second. Be sure to measure and include typical queries and DocumentDB script usage as well.

NOTE

If you have document types which will differ dramatically in terms of size and the number of indexed properties, then record the applicable operation request unit charge associated with each *type* of typical document.

For example:

1. Record the request unit charge of creating (inserting) a typical document.
2. Record the request unit charge of reading a typical document.
3. Record the request unit charge of updating a typical document.
4. Record the request unit charge of typical, common document queries.
5. Record the request unit charge of any custom scripts (stored procedures, triggers, user-defined functions) leveraged by the application
6. Calculate the required request units given the estimated number of operations you anticipate to run each second.

A request unit estimation example

Consider the following ~1KB document:

```

{
  "id": "08259",
  "description": "Cereals ready-to-eat, KELLOGG, KELLOGG'S CRISPIX",
  "tags": [
    {
      "name": "cereals ready-to-eat"
    },
    {
      "name": "kellogg"
    },
    {
      "name": "kellogg's crispix"
    }
  ],
  "version": 1,
  "commonName": "Includes USDA Commodity B855",
  "manufacturerName": "Kellogg, Co.",
  "isFromSurvey": false,
  "foodGroup": "Breakfast Cereals",
  "nutrients": [
    {
      "id": "262",
      "description": "Caffeine",
      "nutritionValue": 0,
      "units": "mg"
    },
    {
      "id": "307",
      "description": "Sodium, Na",
      "nutritionValue": 611,
      "units": "mg"
    },
    {
      "id": "309",
      "description": "Zinc, Zn",
      "nutritionValue": 5.2,
      "units": "mg"
    }
  ],
  "servings": [
    {
      "amount": 1,
      "description": "cup (1 NLEA serving)",
      "weightInGrams": 29
    }
  ]
}

```

NOTE

Documents are minified in DocumentDB, so the system calculated size of the document above is slightly less than 1KB.

The following table shows approximate request unit charges for typical operations on this document (the approximate request unit charge assumes that the account consistency level is set to "Session" and that all documents are automatically indexed):

OPERATION	REQUEST UNIT CHARGE
Create document	~15 RU
Read document	~1 RU

OPERATION	REQUEST UNIT CHARGE
Query document by id	~2.5 RU

Additionally, this table shows approximate request unit charges for typical queries used in the application:

QUERY	REQUEST UNIT CHARGE	# OF RETURNED DOCUMENTS
Select food by id	~2.5 RU	1
Select foods by manufacturer	~7 RU	7
Select by food group and order by weight	~70 RU	100
Select top 10 foods in a food group	~10 RU	10

NOTE

RU charges vary based on the number of documents returned.

With this information, we can estimate the RU requirements for this application given the number of operations and queries we expect per second:

OPERATION/QUERY	ESTIMATED NUMBER PER SECOND	REQUIRED RUS
Create document	10	150
Read document	100	100
Select foods by manufacturer	25	175
Select by food group	10	700
Select top 10	15	150 Total

In this case, we expect an average throughput requirement of 1,275 RU/s. Rounding up to the nearest 100, we would provision 1,300 RU/s for this application's collection.

Exceeding reserved throughput limits

Recall that request unit consumption is evaluated as a rate per second. For applications that exceed the provisioned request unit rate for a collection, requests to that collection will be throttled until the rate drops below the reserved level. When a throttle occurs, the server will preemptively end the request with `RequestRateTooLargeException` (HTTP status code 429) and return the `x-ms-retry-after-ms` header indicating the amount of time, in milliseconds, that the user must wait before reattempting the request.

```
HTTP Status 429
Status Line: RequestRateTooLarge
x-ms-retry-after-ms :100
```

If you are using the .NET Client SDK and LINQ queries, then most of the time you never have to deal with this exception, as the current version of the .NET Client SDK implicitly catches this response, respects the server-

specified retry-after header, and retries the request. Unless your account is being accessed concurrently by multiple clients, the next retry will succeed.

If you have more than one client cumulatively operating above the request rate, the default retry behavior may not suffice, and the client will throw a `DocumentClientException` with status code 429 to the application. In cases such as this, you may consider handling retry behavior and logic in your application's error handling routines or increasing the reserved throughput for the collection.

Next steps

To learn more about reserved throughput with Azure DocumentDB databases, explore these resources:

- [DocumentDB pricing](#)
- [Managing DocumentDB capacity](#)
- [Modeling data in DocumentDB](#)
- [DocumentDB performance levels](#)

To learn more about DocumentDB, see the Azure DocumentDB [documentation](#).

To get started with scale and performance testing with DocumentDB, see [Performance and Scale Testing with Azure DocumentDB](#).

Automate DocumentDB account creation using Azure CLI and Azure Resource Manager templates

11/15/2016 • 16 min to read • [Edit on GitHub](#)

Contributors

mimig • [Ralph Squillace](#) • [Theano Petersen](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [Digvijay Makwana](#) • [tfitzmac](#)
• [Cynthia Nottingham \[MSFT\]](#) • [Jennifer Hubbard](#) • [Andy Pasic](#)

This article shows you how to create an Azure DocumentDB account by using Azure Resource Manager templates or directly with the Azure Command-Line Interface (CLI). To create a DocumentDB account using the Azure portal, see [Create a DocumentDB database account using the Azure portal](#).

DocumentDB database accounts are currently the only DocumentDB resource that can be created using Resource Manager templates and the Azure CLI.

Getting ready

Before you can use the Azure CLI with Azure resource groups, you need to have the right Azure CLI version and an Azure account. If you don't have the Azure CLI, [install it](#).

Update your Azure CLI version

At the command prompt, type `azure --version` to see whether you have already installed version 0.10.4 or later. You may be prompted to participate in Microsoft Azure CLI data collection at this step, and can select y or n to opt-in or opt-out.

```
azure --version
0.10.4 (node: 4.2.4)
```

If your version is not 0.10.4 or later, you need to either [install the Azure CLI](#) or update by using one of the native installers, or through `npm` by typing `npm update -g azure-cli` to update or `npm install -g azure-cli` to install.

Set your Azure account and subscription

If you don't already have an Azure subscription but you do have a Visual Studio subscription, you can activate your [Visual Studio subscriber benefits](#). Or you can sign up for a [free trial](#).

You need to have a work or school account or a Microsoft account identity to use Azure resource management templates. If you have one of these accounts, type the following command:

```
azure login
```

Which produces the following output:

```
info:    Executing command login
|info:    To sign in, use a web browser to open the page https://aka.ms/devicelogin.
Enter the code E1A2B3C4D to authenticate.
```

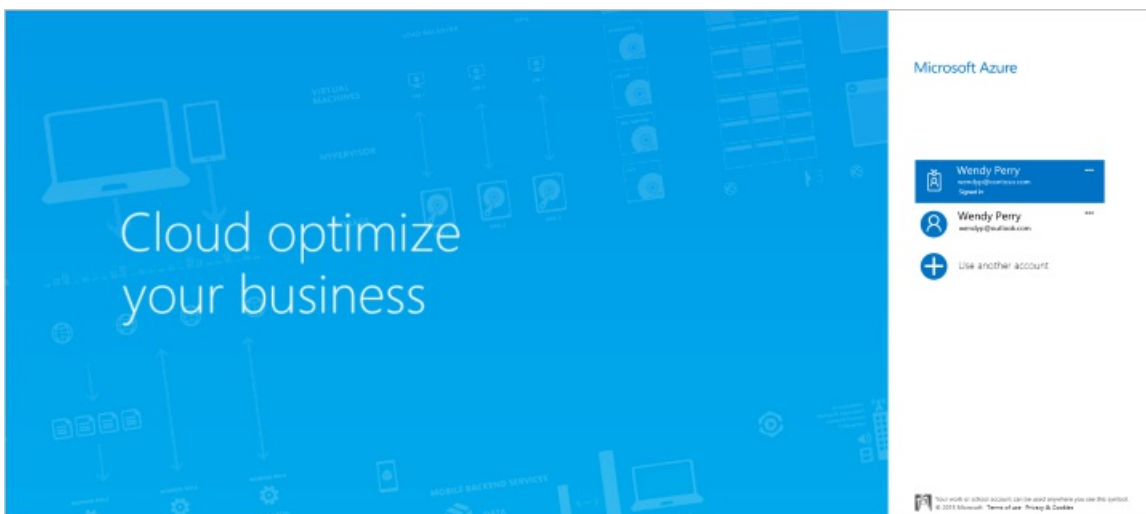
NOTE

If you don't have an Azure account, you see an error message indicating that you need a different type of account. To create one from your current Azure account, see [Creating a work or school identity in Azure Active Directory](#).

Open <https://aka.ms/devicelogin> in a browser and enter the code provided in the command output.



Once you've entered the code, select the identity you want to use in the browser and provide your user name and password if needed.



You receive the following confirmation screen when you're successfully logged in, and you can then close the browser window.



The command shell also provides the following output:

```
/info:    Added subscription Visual Studio Ultimate with MSDN
info:    Setting subscription "Visual Studio Ultimate with MSDN" as default
+
info:    login command OK
```

In addition to the interactive login method described here, there are additional Azure CLI login methods available. For more information about the other methods and information about handling multiple subscriptions, see [Connect to an Azure subscription from the Azure Command-Line Interface \(Azure CLI\)](#).

Switch to the Azure CLI resource group mode

By default, the Azure CLI starts in the service management mode (**asm** mode). Type the following to switch to resource group mode.

```
azure config mode arm
```

Which provides the following output:

```
info:    Executing command config mode
info:    New mode is arm
info:    config mode command OK
```

If needed, you can switch back to the default set of commands by typing `azure config mode asm`.

Create or retrieve your resource group

To create a DocumentDB account, you first need a resource group. If you already know the name of the resource group that you'd like to use, then skip to [Step 2](#).

To review a list of all your current resource groups, run the following command and take note of the resource group name you'd like to use:

```
azure group list
```

To create a resource group, run the following command, specify the name of the new resource group to create, and the region in which to create the resource group:

```
azure group create <resourcegroupname> <resourcegrouplocation>
```

- `<resourcegroupname>` can only use alphanumeric characters, periods, underscores, the '-' character, and parenthesis and cannot end in a period.
- `<resourcegrouplocation>` must be one of the regions in which DocumentDB is generally available. The current list of regions is provided on the [Azure Regions page](#).

Example input:

```
azure group create new_res_group westus
```

Which produces the following output:

```
info:    Executing command group create
+ Getting resource group new_res_group
+ Creating resource group new_res_group
info:    Created resource group new_res_group
data:    Id:                /subscriptions/xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/resourceGroups/new_res_group
data:    Name:                new_res_group
data:    Location:            westus
data:    Provisioning State:  Succeeded
data:    Tags: null
data:
info:    group create command OK
```

If you encounter errors, see [Troubleshooting](#).

Understanding Resource Manager templates and resource groups

Most applications are built from a combination of different resource types (such as one or more DocumentDB account, storage accounts, a virtual network, or a content delivery network). The default Azure service management API and the Azure portal represented these items by using a service-by-service approach. This approach requires you to deploy and manage the individual services individually (or find other tools that do so), and not as a single logical unit of deployment.

Azure Resource Manager templates make it possible for you to deploy and manage these different resources as one logical deployment unit in a declarative fashion. Instead of imperatively telling Azure what to deploy one command after another, you describe your entire deployment in a JSON file -- all the resources and associated configuration and deployment parameters -- and tell Azure to deploy those resources as one group.

You can learn lots more about Azure resource groups and what they can do for you in the [Azure Resource Manager overview](#). If you're interested in authoring templates, see [Authoring Azure Resource Manager templates](#).

Task: Create a Single Region DocumentDB account

Use the instructions in this section to create a Single Region DocumentDB account. This can be accomplished using Azure CLI with or without Resource Manager templates.

Create a Single Region DocumentDB account using Azure CLI without Resource Manager templates

Create a DocumentDB account in the new or existing resource group by entering the following command at the command prompt:

TIP

If you run this command in Azure PowerShell or Windows PowerShell you receive an error about an unexpected token. Instead, run this command at the Windows Command Prompt.

```
azure resource create -g <resourcegroupname> -n <databaseaccountname> -r "Microsoft.DocumentDB/databaseAccounts"
-o 2015-04-08 -l <resourcegrouplocation> -p "{\"databaseAccountOfferType\":\"Standard\",\"locations\":[\"
{\"locationName\":\"<databaseaccountlocation>\",\"failoverPriority\":\"<failoverPriority>\"}]}"
```

- `<resourcegroupname>` can only use alphanumeric characters, periods, underscores, the '-' character, and parenthesis and cannot end in a period.
- `<resourcegrouplocation>` is the region of the current resource group.
- `<databaseaccountname>` can only use lowercase letters, numbers, the '-' character, and must be between 3 and 50 characters.
- `<databaseaccountlocation>` must be one of the regions in which DocumentDB is generally available. The current list of regions is provided on the [Azure Regions page](#).

Example input:

```
azure resource create -g new_res_group -n samplecliacct -r "Microsoft.DocumentDB/databaseAccounts" -o 2015-04-08
-l westus -p "{\"databaseAccountOfferType\":\"Standard\",\"locations\":["
{\"locationName\":\"westus\",\"failoverPriority\":\"0\"}]}"
```

Which produces the following output as your new account is provisioned:

```
info:    Executing command resource create
+ Getting resource samplecliacct
+ Creating resource samplecliacct
info:    Resource samplecliacct is updated
data:
data:    Id:          /subscriptions/xxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxx/resourceGroups/new_res_group/providers/Microsoft.DocumentDB/databaseAccounts/samplecliacct
data:    Name:         samplecliacct
data:    Type:          Microsoft.DocumentDB/databaseAccounts
data:    Parent:
data:    Location:     West US
data:    Tags:
data:
info:    resource create command OK
```

If you encounter errors, see [Troubleshooting](#).

After the command returns, the account will be in the **Creating** state for a few minutes, before it changes to the **Online** state in which it is ready for use. You can check on the status of the account in the [Azure portal](#), on the **DocumentDB Accounts** blade.

Create a Single Region DocumentDB account using Azure CLI with Resource Manager templates

The instructions in this section describe how to create a DocumentDB account with an Azure Resource Manager template and an optional parameters file, both of which are JSON files. Using a template enables you to describe exactly what you want and repeat it without errors.

Create a local template file with the following content. Name the file `azuredeploy.json`.

```
{
  "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "databaseAccountName": {
      "type": "string"
    },
    "locationName1": {
      "type": "string"
    }
  },
  "variables": {},
  "resources": [
    {
      "apiVersion": "2015-04-08",
      "type": "Microsoft.DocumentDb/databaseAccounts",
      "name": "[parameters('databaseAccountName')]",
      "location": "[resourceGroup().location]",
      "properties": {
        "databaseAccountOfferType": "Standard",
        "locations": [
          {
            "failoverPriority": 0,
            "locationName": "[parameters('locationName1')]"
          }
        ]
      }
    }
  ]
}
```

The failoverPriority must be set to 0 since this is a single region account. A failoverPriority of 0 indicates that this region be kept as the [write region for the DocumentDB account](#). You can either enter the value at the command line, or create a parameter file to specify the value.

To create a parameters file, copy the following content into a new file and name the file azuredeploy.parameters.json. If you plan on specifying the database account name at the command prompt, you can continue without creating this file.

```
{
  "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "databaseAccountName": {
      "value": "samplearmacct"
    },
    "locationName1": {
      "value": "westus"
    }
  }
}
```

In the azuredeploy.parameters.json file, update the value field of "samplearmacct" to the database name you'd like to use, then save the file. "databaseAccountName" can only use lowercase letters, numbers, the '-' character, and must be between 3 and 50 characters. Update the value field of "locationName1" to the region where you would like to create the DocumentDB account.

To create a DocumentDB account in your resource group, run the following command and provide the path to the template file, the path to the parameter file or the parameter value, the name of the resource group in which to deploy, and a deployment name (-n is optional).

To use a parameter file:

```
azure group deployment create -f <PathToTemplate> -e <PathToParameterFile> -g <resourcegroupname> -n <deploymentname>
```

- `<PathToTemplate>` is the path to the `azuredeploy.json` file created in step 1. If your path name has spaces in it, put double quotes around this parameter.
- `<PathToParameterFile>` is the path to the `azuredeploy.parameters.json` file created in step 1. If your path name has spaces in it, put double quotes around this parameter.
- `<resourcegroupname>` is the name of the existing resource group in which to add a DocumentDB database account.
- `<deploymentname>` is the optional name of the deployment.

Example input:

```
azure group deployment create -f azuredeploy.json -e azuredeploy.parameters.json -g new_res_group -n azuredeploy
```

OR to specify the database account name parameter without a parameter file, and instead get prompted for the value, run the following command:

```
azure group deployment create -f <PathToTemplate> -g <resourcegroupname> -n <deploymentname>
```

Example input which shows the prompt and entry for a database account named `samplearmacct`:

```
azure group deployment create -f azuredeploy.json -g new_res_group -n azuredeploy
info:   Executing command group deployment create
info:   Supply values for the following parameters
databaseAccountName: samplearmacct
```

As the account is provisioned, you receive the following information:

```
info:   Executing command group deployment create
+ Initializing template configurations and parameters
+ Creating a deployment
info:   Created template deployment "azuredeploy"
+ Waiting for deployment to complete
+
+
info:   Resource 'new_res_group' of type 'Microsoft.DocumentDb/databaseAccounts' provisioning status is Running
+
info:   Resource 'new_res_group' of type 'Microsoft.DocumentDb/databaseAccounts' provisioning status is Succeeded
data:   DeploymentName      : azuredeploy
data:   ResourceGroupName    : new_res_group
data:   ProvisioningState     : Succeeded
data:   Timestamp            : 2015-11-30T18:50:23.6300288Z
data:   Mode                 : Incremental
data:   CorrelationId        : 4a5d4049-c494-4053-bad4-cc804d454700
data:   DeploymentParameters :
data:   Name                 Type   Value
data:   -----
data:   databaseAccountName  String samplearmacct
data:   locationName1        String westus
info:   group deployment create command OK
```

If you encounter errors, see [Troubleshooting](#).

After the command returns, the account will be in the **Creating** state for a few minutes, before it changes to the **Online** state in which it is ready for use. You can check on the status of the account in the [Azure portal](#), on the **DocumentDB Accounts** blade.

Task: Create a multi-region DocumentDB account

DocumentDB has the capability to [distribute your data globally](#) across various [Azure regions](#). When creating a DocumentDB account, the regions in which you would like the service to exist can be specified. Use the instructions in this section to create a multi-region DocumentDB account. This can be accomplished using Azure CLI with or without Resource Manager templates.

Create a multi-region DocumentDB account using Azure CLI without Resource Manager templates

Create a DocumentDB account in the new or existing resource group by entering the following command at the command prompt:

TIP

If you run this command in Azure PowerShell or Windows PowerShell you receive an error about an unexpected token. Instead, run this command at the Windows Command Prompt.

```
azure resource create -g <resourcegroupname> -n <databaseaccountname> -r "Microsoft.DocumentDB/databaseAccounts"
-o 2015-04-08 -l <resourcegrouplocation> -p "{\"databaseAccountOfferType\":\"Standard\",\"locations\":[\"
{\\\"locationName\\\":\\\"<databaseaccountlocation1>\\\",\\\"failoverPriority\\\":\\\"<failoverPriority1>\\\"},
{\\\"locationName\\\":\\\"<databaseaccountlocation2>\\\",\\\"failoverPriority\\\":\\\"<failoverPriority2>\\\"}]}"
```

- `<resourcegroupname>` can only use alphanumeric characters, periods, underscores, the '-' character, and parenthesis and cannot end in a period.
- `<resourcegrouplocation>` is the region of the current resource group.
- `<databaseaccountname>` can only use lowercase letters, numbers, the '-' character, and must be between 3 and 50 characters.
- `<databaseaccountlocation1>` and `<databaseaccountlocation2>` must be regions in which DocumentDB is generally available. The current list of regions is provided on the [Azure Regions page](#).

Example input:

```
azure resource create -g new_res_group -n samplecliacct -r "Microsoft.DocumentDB/databaseAccounts" -o 2015-04-08
-l westus -p "{\"databaseAccountOfferType\":\"Standard\",\"locations\":[\"
{\\\"locationName\\\":\\\"westus\\\",\\\"failoverPriority\\\":\\\"0\\\"},
{\\\"locationName\\\":\\\"eastus\\\",\\\"failoverPriority\\\":\\\"1\\\"}]}"
```

Which produces the following output as your new account is provisioned:

```
info:    Executing command resource create
+ Getting resource samplecliacct
+ Creating resource samplecliacct
info:    Resource samplecliacct is updated
data:
data:    Id:          /subscriptions/xxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx/resourceGroups/new_res_group/providers/Microsoft.DocumentDB/databaseAccounts/samplecliacct
data:    Name:         samplecliacct
data:    Type:          Microsoft.DocumentDB/databaseAccounts
data:    Parent:
data:    Location:     West US
data:    Tags:
data:
info:    resource create command OK
```

If you encounter errors, see [Troubleshooting](#).

After the command returns, the account will be in the **Creating** state for a few minutes, before it changes to the **Online** state in which it is ready for use. You can check on the status of the account in the [Azure portal](#), on the

DocumentDB Accounts blade.

Create a multi-region DocumentDB account using Azure CLI with Resource Manager templates

The instructions in this section describe how to create a DocumentDB account with an Azure Resource Manager template and an optional parameters file, both of which are JSON files. Using a template enables you to describe exactly what you want and repeat it without errors.

Create a local template file with the following content. Name the file `azuredeploy.json`.

```
{
  "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "databaseAccountName": {
      "type": "string"
    },
    "locationName1": {
      "type": "string"
    },
    "locationName2": {
      "type": "string"
    }
  },
  "variables": {},
  "resources": [
    {
      "apiVersion": "2015-04-08",
      "type": "Microsoft.DocumentDb/databaseAccounts",
      "name": "[parameters('databaseAccountName')]",
      "location": "[resourceGroup().location]",
      "properties": {
        "databaseAccountOfferType": "Standard",
        "locations": [
          {
            "failoverPriority": 0,
            "locationName": "[parameters('locationName1')]"
          },
          {
            "failoverPriority": 1,
            "locationName": "[parameters('locationName2')]"
          }
        ]
      }
    }
  ]
}
```

The preceding template file can be used to create a DocumentDB account with two regions. To create the account with more regions, add it to the "locations" array and add the corresponding parameters.

One of the regions must have a `failoverPriority` value of 0 to indicate that this region be kept as the [write region for the DocumentDB account](#). The failover priority values must be unique among the locations and the highest failover priority value must be less than the total number of regions. You can either enter the value at the command line, or create a parameter file to specify the value.

To create a parameters file, copy the following content into a new file and name the file `azuredeploy.parameters.json`. If you plan on specifying the database account name at the command prompt, you can continue without creating this file.

```
{
  "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "databaseAccountName": {
      "value": "samplearmacct"
    },
    "locationName1": {
      "value": "westus"
    },
    "locationName2": {
      "value": "eastus"
    }
  }
}
```

In the `azuredeploy.parameters.json` file, update the value field of `"samplearmacct"` to the database name you'd like to use, then save the file. `"databaseAccountName"` can only use lowercase letters, numbers, the '-' character, and must be between 3 and 50 characters. Update the value field of `"locationName1"` and `"locationName2"` to the region where you would like to create the DocumentDB account.

To create a DocumentDB account in your resource group, run the following command and provide the path to the template file, the path to the parameter file or the parameter value, the name of the resource group in which to deploy, and a deployment name (-n is optional).

To use a parameter file:

```
azure group deployment create -f <PathToTemplate> -e <PathToParameterFile> -g <resourcegroupname> -n
<deploymentname>
```

- `<PathToTemplate>` is the path to the `azuredeploy.json` file created in step 1. If your path name has spaces in it, put double quotes around this parameter.
- `<PathToParameterFile>` is the path to the `azuredeploy.parameters.json` file created in step 1. If your path name has spaces in it, put double quotes around this parameter.
- `<resourcegroupname>` is the name of the existing resource group in which to add a DocumentDB database account.
- `<deploymentname>` is the optional name of the deployment.

Example input:

```
azure group deployment create -f azuredeploy.json -e azuredeploy.parameters.json -g new_res_group -n azuredeploy
```

OR to specify the database account name parameter without a parameter file, and instead get prompted for the value, run the following command:

```
azure group deployment create -f <PathToTemplate> -g <resourcegroupname> -n <deploymentname>
```

Example input, which shows the prompt and entry for a database account named `samplearmacct`:

```
azure group deployment create -f azuredeploy.json -g new_res_group -n azuredeploy
info: Executing command group deployment create
info: Supply values for the following parameters
databaseAccountName: samplearmacct
```

As the account is provisioned, you receive the following information:

```

info:    Executing command group deployment create
+ Initializing template configurations and parameters
+ Creating a deployment
info:    Created template deployment "azuredeploy"
+ Waiting for deployment to complete
+
+
info:    Resource 'new_res_group' of type 'Microsoft.DocumentDb/databaseAccounts' provisioning status is Running
+
info:    Resource 'new_res_group' of type 'Microsoft.DocumentDb/databaseAccounts' provisioning status is Succeeded
data:    DeploymentName      : azuredeploy
data:    ResourceGroupName     : new_res_group
data:    ProvisioningState      : Succeeded
data:    Timestamp              : 2015-11-30T18:50:23.6300288Z
data:    Mode                   : Incremental
data:    CorrelationId          : 4a5d4049-c494-4053-bad4-cc804d454700
data:    DeploymentParameters :
data:    Name                   Type    Value
data:    -----
data:    databaseAccountName    String samplearmacct
data:    locationName1          String westus
data:    locationName2          String eastus
info:    group deployment create command OK

```

If you encounter errors, see [Troubleshooting](#).

After the command returns, the account will be in the **Creating** state for a few minutes, before it changes to the **Online** state in which it is ready for use. You can check on the status of the account in the [Azure portal](#), on the **DocumentDB Accounts** blade.

Troubleshooting

If you receive errors like `Deployment provisioning state was not successful` while creating your resource group or database account, you have a few troubleshooting options.

NOTE

Providing incorrect characters in the database account name or providing a location in which DocumentDB is not available will cause deployment errors. Database account names can only use lowercase letters, numbers, the '-' character, and must be between 3 and 50 characters. All valid database account locations are listed on the [Azure Regions page](#).

- If your output contains the following `Error information has been recorded to C:\Users\wendy\.azure\azure.err`, then review the error info in the azure.err file.
- You may find useful info in the log file for the resource group. To view the log file, run the following command:

```
azure group log show <resourcegroupname> --last-deployment
```

Example input:

```
azure group log show new_res_group --last-deployment
```

Then see [Troubleshooting resource group deployments in Azure](#) for additional information.

- Error information is also available in the Azure portal as shown in the following screenshot. To navigate to the error info: click Resource Groups in the Jumpbar, select the Resource Group that had the error, then in the Essentials area of the Resource group blade click the date of the Last Deployment, then in the

[illegible]

Now that you have a DocumentDB account, the next step is to create a DocumentDB database. You can create a database by using one of the following:

- After creating your database, you need to [add one or more collections](#) to the database, then [add documents](#) to the collections.

To learn more about DocumentDB, explore these resources:

- For more templates you can use, see [Azure Quickstart templates](#).

DocumentDB firewall support

11/15/2016 • 4 min to read • [Edit on GitHub](#)

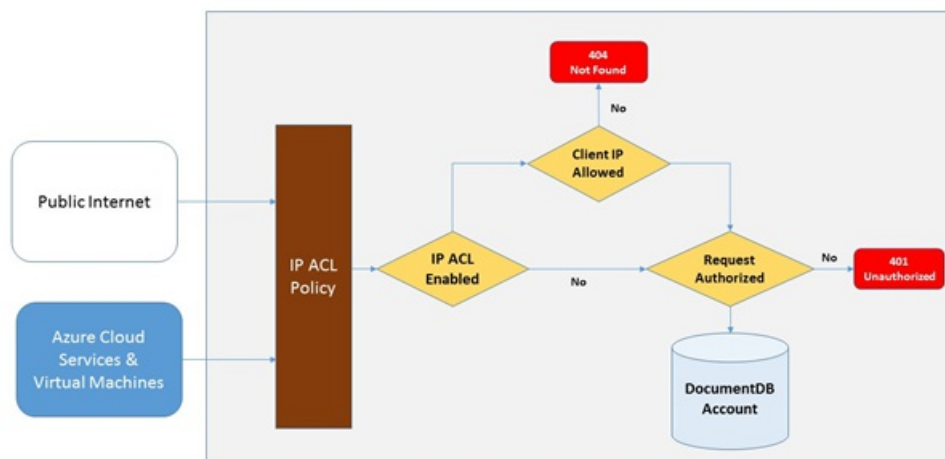
Contributors

[Ankur Shah](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [mimig](#)

To secure data stored in an Azure DocumentDB database account, DocumentDB has provided support for a secret based [authorization model](#) that utilizes a strong Hash-based message authentication code (HMAC). Now, in addition to the secret based authorization model, DocumentDB supports policy driven IP-based access controls for inbound firewall support. This model is very similar to the firewall rules of a traditional database system and provides an additional level of security to the DocumentDB database account. With this model, you can now configure a DocumentDB database account to be accessible only from an approved set of machines and/or cloud services. Access to DocumentDB resources from these approved sets of machines and services still require the caller to present a valid authorization token.

IP access control overview

By default, a DocumentDB database account is accessible from public internet as long as the request is accompanied by a valid authorization token. To configure IP policy-based access control, the user must provide the set of IP addresses or IP address ranges in CIDR form to be included as the allowed list of client IPs for a given database account. Once this configuration is applied, all requests originating from machines outside this allowed list will be blocked by the server. The connection processing flow for the IP-based access control is described in the following diagram.



Connections from cloud services

In Azure, cloud services are a very common way for hosting middle tier service logic using DocumentDB. To enable access to a DocumentDB database account from a cloud service, the public IP address of the cloud service must be added to the allowed list of IP addresses associated with your DocumentDB database account by [contacting Azure support](#). This ensures that all role instances of cloud services have access to your DocumentDB database account. You can retrieve IP addresses for your cloud services in the Azure portal, as shown in the following screenshot.

Search (Ctrl+ /)

Overview

Activity log

Access control (IAM)

SETTINGS

Antimalware

Certificates

Configuration

Extensions

Remote Desktop

Scale

Properties

Locks

MONITORING

Alert rules

CURRENT SLOT

Production

STATUS

Running

SITE URL

http://telcoworkernortheurope.cloudapp....

NAME

648e876725b946ec8f14ebadd4b82f92

LABEL

ContosoTelcoWorker - 9/18/2016 4:41:35...

DEPLOYMENT ID

d00bc8996e5240a793692432d0137ef1

PUBLIC IP ADDRESSES

13.74.156.31

INPUT ENDPOINTS

TelcoMetricWorker: 13.74.156.31:3389

When you scale out your cloud service by adding additional role instance(s), those new instances will automatically have access to the DocumentDB database account since they are part of the same cloud service.

Connections from virtual machines

[Virtual machines](#) or [virtual machine scale sets](#) can also be used to host middle tier services using DocumentDB. To configure the DocumentDB database account to allow access from virtual machines, public IP addresses of virtual machine and/or virtual machine scale set must be configured as one of the allowed IP addresses for your DocumentDB database account by [contacting Azure support](#). You can retrieve IP addresses for virtual machines in the Azure portal, as shown in the following screenshot.

Search (Ctrl+ /)

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

SETTINGS

Availability set

Disks

STATUS

Running

COMPUTER NAME

cache

PUBLIC IP ADDRESS/DNS NAME LABEL

13.75.145.220/<none>

PRIVATE IP ADDRESS

10.0.0.4

OPERATING SYSTEM

Windows

When you add additional virtual machine instances to the group, they are automatically provided access to your DocumentDB database account.

Connections from the internet

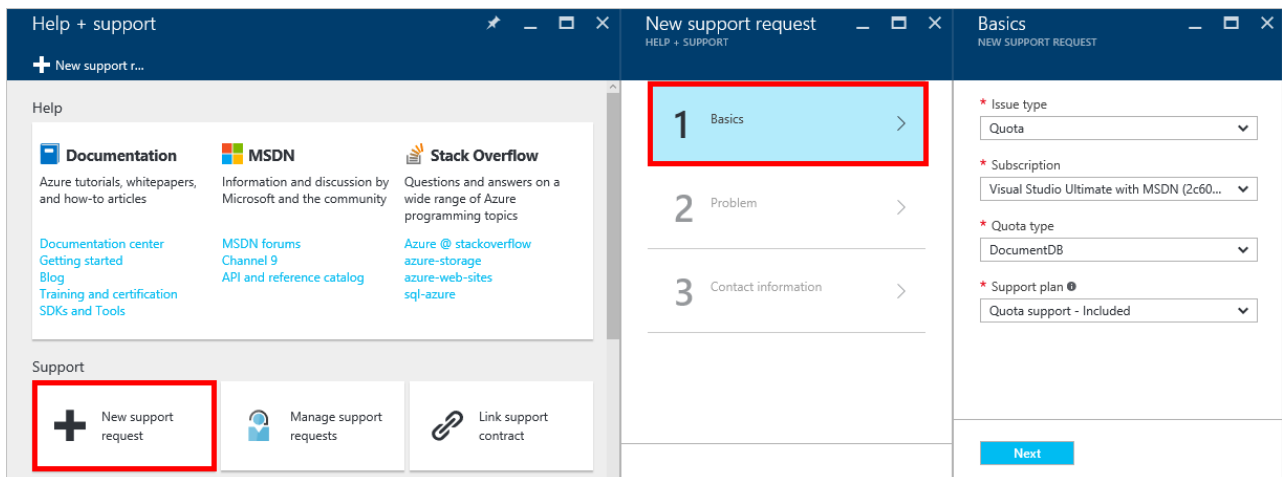
When you access a DocumentDB database account from a computer on the internet, the client IP address or IP address range of the machine must be added to the allowed list of IP address for the DocumentDB database account.

Configuring the IP access control policy

Use the Azure portal to file a request with [Azure Support](#) to enable the IP access control policy on your database account.

1. In the [Help + support](#) blade, select **New support request**.
2. In the **New support request** blade, select **Basics**.
3. In the **Basics** blade, select the following:
 - **Issue type:** Quota
 - **Subscription:** The subscription associated with the account in which to add the IP access control policy.
 - **Quota type:** DocumentDB
 - **Support plan:** Quota Support - Included.
4. In the **Problem** blade, do the following:
 - **Severity:** Select C - Minimal impact
 - **Details:** Copy the following text into the box, and include your account name/s and IP address/es: "I would like to enable firewall support for my DocumentDB database account. Database account: *Include account name/s*. Allowed IP address/Ranges: *Include IP address/range in CIDR format, for example 13.91.6.132, 13.91.6.1/24*."
 - Click **Next**.
5. In the **Contact information** blade, fill in your contact details and click **Create**.


Once your request is received, IP access control should be enabled within 24 hours. You will be notified once the request is complete.



Problem
NEW SUPPORT REQUEST

* Severity ⓘ
C - Minimal impact ▼

* Details
I would like to enable firewall support for my DocumentDB database account. ✓
Database account: <Include account name/s>. Allowed IP address/Ranges: <Include IP address/range in CIDR format, for example 13.91.6.132, 13.91.6.1/24>.

File upload ⓘ
Select a file 

Next

Troubleshooting the IP access control policy

Portal operations

By enabling an IP access control policy for your DocumentDB database account, all access to your DocumentDB database account from machines outside the configured allowed list of IP address ranges are blocked. By virtue of this model, browsing the data plane operation from the portal will also be blocked to ensure the integrity of access control.

SDK & Rest API

For security reasons, access via SDK or REST API from machines not on the allowed list will return a generic 404 Not Found response with no additional details. Please verify the IP allowed list configured for your DocumentDB database account to ensure the correct policy configuration is applied to your DocumentDB database account.

Next steps

For information about network related performance tips, see [Performance tips](#).

Supercharge your DocumentDB account

11/15/2016 • 1 min to read • [Edit on GitHub](#)

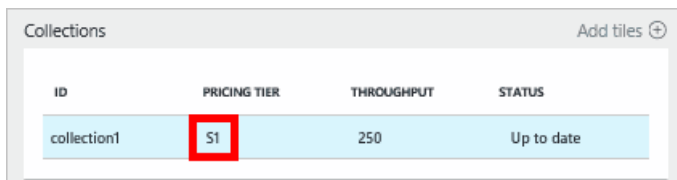
Contributors

[mimig](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#)

Follow these steps to take advantage of increased throughput for your Azure DocumentDB S1 account. With little to no additional cost, you can increase the throughput of your existing S1 account from 250 [RU/s](#) to 400 RU/s, or more!

Change to user-defined performance in the Azure portal

1. In your browser, navigate to the [Azure portal](#).
2. Click **Browse** -> **DocumentDB (NoSQL)**, then select the DocumentDB account to modify.
3. In the **Databases** lens, select the database to modify, and then in the **Database** blade, select the collection with the S1 pricing tier.



ID	PRICING TIER	THROUGHPUT	STATUS
collection1	S1	250	Up to date

4. In the **Collection** blade, click **More**, and then click **Settings**.
5. In the **Settings** blade, click **Pricing Tier** and notice that the monthly cost estimate for each plan is displayed. In the **Choose your pricing tier** blade, click **Standard**, and then click **Select** to save your change.

Settings

PRICING TIER ⓘ

S1

>

THROUGHPUT (RU/s) ⓘ

250

DEFAULT STORAGE CAPACITY ⓘ

10 GB

INDEXING POLICY ⓘ

Default

>

TIME TO LIVE ⓘ

On

On (no default)

Off

Pricing Summary (Monthly Estimated)

25 USD

OK

Choose your pricing tier

DocumentDB supports a simple and flexible pricing model. Collections are billed based on the volume of data stored and the throughput provisioned.

★ Recommended

View all

STANDARD	S1	S2
6.00 USD per 100 RU/s	250 RUs	1K RUs
0.25 USD per GB used		
<div>Single Partition</div> <div>Up to 10k RU/s & 10GB</div>	<div>Single Partition</div>	<div>Single Partition</div>
<div>Partitioned</div> <div>Unlimited RU/s and storage</div>	<div>10 GB Storage</div>	<div>10 GB Storage</div>
<div>99.99% Availability SLA</div>	<div>99.99% Availability SLA</div>	<div>99.99% Availability SLA</div>
24.00	25.00	50.00
STARTING COST (USD)/MONTH	USD/MONTH (ESTIMATED)	USD/MONTH (ESTIMATED)
S3		
2.5K RUs		
<div>Single Partition</div>		
<div>10 GB Storage</div>		
<div>99.99% Availability SLA</div>		
100.00		
USD/MONTH (ESTIMATED)		

Select

6. Back in the **Settings** blade, the **Pricing Tier** is changed to **Standard** and the **Throughput (RU/s)** box is displayed with a default value of 400. Click **OK** to save your changes.

NOTE

You can set the throughput between 400 and 10,000 [Request units/second \(RU/s\)](#). The **Pricing Summary** at the bottom of the page updates automatically to provide an estimate of the monthly cost.

Settings

Collections are containers for JSON documents. These are billable entities [Learn more](#)

PRICING TIER ⓘ

Standard

* THROUGHPUT (RU/s) ⓘ

400 ✓

Between 400 and 10,000 RUs

DEFAULT STORAGE CAPACITY ⓘ

10 GB

INDEXING POLICY ⓘ

Default

TIME TO LIVE ⓘ

On

On (no default)

Off

Pricing Summary (Monthly Estimated)

THROUGHPUT (6.00 USD/100 RU/s)

24.00 USD

STORAGE (0.25 USD/GB)

0.00 USD

OK

7. Back on the **Database** blade, you can verify the supercharged throughput of the collection.

Collections Add tiles +			
ID	PRICING TIER	THROUGHPUT	STATUS
collection1	Standard	400	Up to date

For more information about the changes related to user-defined and pre-defined throughput, see the blog post [DocumentDB: Everything you need to know about using the new pricing options](#).

Next steps

If you determine that you need more throughput (greater than 10,000 RU/s) or more storage (greater than 10GB) you can create a partitioned collection. To create a partitioned collection, see [Create a collection](#).

SQL query and SQL syntax in DocumentDB

11/22/2016 • 50 min to read • [Edit on GitHub](#)

Contributors

arramac • mimig • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Dene Hager • Ross McAllister
• Jennifer Hubbard

Microsoft Azure DocumentDB supports querying documents using SQL (Structured Query Language) as a JSON query language. DocumentDB is truly schema-free. By virtue of its commitment to the JSON data model directly within the database engine, it provides automatic indexing of JSON documents without requiring explicit schema or creation of secondary indexes.

While designing the query language for DocumentDB we had two goals in mind:

- Instead of inventing a new JSON query language, we wanted to support SQL. SQL is one of the most familiar and popular query languages. DocumentDB SQL provides a formal programming model for rich queries over JSON documents.
- As a JSON document database capable of executing JavaScript directly in the database engine, we wanted to use JavaScript's programming model as the foundation for our query language. The DocumentDB SQL is rooted in JavaScript's type system, expression evaluation, and function invocation. This in-turn provides a natural programming model for relational projections, hierarchical navigation across JSON documents, self joins, spatial queries, and invocation of user defined functions (UDFs) written entirely in JavaScript, among other features.

We believe that these capabilities are key to reducing the friction between the application and the database and are crucial for developer productivity.

We recommend getting started by watching the following video, where Aravind Ramachandran shows DocumentDB's querying capabilities, and by visiting our [Query Playground](#), where you can try out DocumentDB and run SQL queries against our dataset.

Then, return to this article, where we'll start with a SQL query tutorial that walks you through some simple

JSON documents and SQL commands.

Getting started with SQL commands in DocumentDB

To see DocumentDB SQL at work, let's begin with a few simple JSON documents and walk through some simple queries against it. Consider these two JSON documents about two families. Note that with DocumentDB, we do not need to create any schemas or secondary indices explicitly. We simply need to insert the JSON documents to a DocumentDB collection and subsequently query. Here we have a simple JSON document for the Andersen family, the parents, children (and their pets), address and registration information. The document has strings, numbers, booleans, arrays and nested properties.

Document

```
{
  "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
}
```

Here's a second document with one subtle difference – `givenName` and `familyName` are used instead of `firstName` and `lastName`.

Document

```
{
  "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
}
```

Now let's try a few queries against this data to understand some of the key aspects of DocumentDB SQL. For example, the following query will return the documents where the id field matches `AndersenFamily`. Since it's a `SELECT *`, the output of the query is the complete JSON document:

Query

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

Results

```
[{
  "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
}]
```

Now consider the case where we need to reformat the JSON output in a different shape. This query projects a new JSON object with two selected fields, Name and City, when the address' city has the same name as the state. In this case, "NY, NY" matches.

Query

```
SELECT {"Name":f.id, "City":f.address.city} AS Family
FROM Families f
WHERE f.address.city = f.address.state
```

Results

```
[{
  "Family": {
    "Name": "WakefieldFamily",
    "City": "NY"
  }
}]
```

The next query returns all the given names of children in the family whose id matches `WakefieldFamily` ordered by the city of residence.

Query

```
SELECT c.givenName
FROM Families f
JOIN c IN f.children
WHERE f.id = 'WakefieldFamily'
ORDER BY f.address.city ASC
```

Results

```
[
  { "givenName": "Jesse" },
  { "givenName": "Lisa"}
]
```

We would like to draw attention to a few noteworthy aspects of the DocumentDB query language through the examples we've seen so far:

- Since DocumentDB SQL works on JSON values, it deals with tree shaped entities instead of rows and columns. Therefore, the language lets you refer to nodes of the tree at any arbitrary depth, like `Node1.Node2.Node3... .NodeN`, similar to relational SQL referring to the two part reference of `<table>.<column>`.
- The structured query language works with schema-less data. Therefore, the type system needs to be bound dynamically. The same expression could yield different types on different documents. The result of a query is a valid JSON value, but is not guaranteed to be of a fixed schema.
- DocumentDB only supports strict JSON documents. This means the type system and expressions are restricted to deal only with JSON types. Please refer to the [JSON specification](#) for more details.
- A DocumentDB collection is a schema-free container of JSON documents. The relations in data entities within and across documents in a collection are implicitly captured by containment and not by primary key and foreign key relations. This is an important aspect worth pointing out in light of the intra-document joins discussed later in this article.

DocumentDB indexing

Before we get into the DocumentDB SQL syntax, it is worth exploring the indexing design in DocumentDB.

The purpose of database indexes is to serve queries in their various forms and shapes with minimum resource consumption (like CPU and input/output) while providing good throughput and low latency. Often, the choice of the right index for querying a database requires much planning and experimentation. This approach poses a challenge for schema-less databases where the data doesn't conform to a strict schema and evolves rapidly.

Therefore, when we designed the DocumentDB indexing subsystem, we set the following goals:

- Index documents without requiring schema: The indexing subsystem does not require any schema information or make any assumptions about schema of the documents.
- Support for efficient, rich hierarchical, and relational queries: The index supports the DocumentDB query language efficiently, including support for hierarchical and relational projections.
- Support for consistent queries in face of a sustained volume of writes: For high write throughput workloads with consistent queries, the index is updated incrementally, efficiently, and online in the face of a sustained volume of writes. The consistent index update is crucial to serve the queries at the consistency level in which the user configured the document service.
- Support for multi-tenancy: Given the reservation based model for resource governance across tenants, index updates are performed within the budget of system resources (CPU, memory, and input/output operations per second) allocated per replica.
- Storage efficiency: For cost effectiveness, the on-disk storage overhead of the index is bounded and predictable. This is crucial because DocumentDB allows the developer to make cost based tradeoffs between index overhead in relation to the query performance.

Refer to the [DocumentDB samples](#) on MSDN for samples showing how to configure the indexing policy for a collection. Let's now get into the details of the DocumentDB SQL syntax.

Basics of a DocumentDB SQL query

Every query consists of a SELECT clause and optional FROM and WHERE clauses per ANSI-SQL standards. Typically, for each query, the source in the FROM clause is enumerated. Then the filter in the WHERE clause is applied on the source to retrieve a subset of JSON documents. Finally, the SELECT clause is used to project the requested JSON values in the select list.

```
SELECT [TOP <top_expression>] <select_list>
[FROM <from_specification>]
[WHERE <filter_condition>]
[ORDER BY <sort_specification>]
```

FROM clause

The `FROM <from_specification>` clause is optional unless the source is filtered or projected later in the query. The purpose of this clause is to specify the data source upon which the query must operate. Commonly the whole collection is the source, but one can specify a subset of the collection instead.

A query like `SELECT * FROM Families` indicates that the entire Families collection is the source over which to enumerate. A special identifier ROOT can be used to represent the collection instead of using the collection name. The following list contains the rules that are enforced per query:

- The collection can be aliased, such as `SELECT f.id FROM Families AS f` or simply `SELECT f.id FROM Families f`. Here `f` is the equivalent of `Families`. `AS` is an optional keyword to alias the identifier.
- Note that once aliased, the original source cannot be bound. For example, `SELECT Families.id FROM Families f` is syntactically invalid since the identifier "Families" cannot be resolved anymore.
- All properties that need to be referenced must be fully qualified. In the absence of strict schema adherence, this is enforced to avoid any ambiguous bindings. Therefore, `SELECT id FROM Families f` is syntactically invalid since the property `id` is not bound.

Sub-documents

The source can also be reduced to a smaller subset. For instance, to enumerating only a sub-tree in each document, the sub-root could then become the source, as shown in the following example.

Query

```
SELECT *
FROM Families.children
```

Results

```
[
  [
    {
      "firstName": "Henriette Thaulow",
      "gender": "female",
      "grade": 5,
      "pets": [
        {
          "givenName": "Fluffy"
        }
      ]
    }
  ],
  [
    {
      "familyName": "Merriam",
      "givenName": "Jesse",
      "gender": "female",
      "grade": 1
    },
    {
      "familyName": "Miller",
      "givenName": "Lisa",
      "gender": "female",
      "grade": 8
    }
  ]
]
```

While the above example used an array as the source, an object could also be used as the source, which is what's shown in the following example. Any valid JSON value (not undefined) that can be found in the source will be considered for inclusion in the result of the query. If some families don't have an `address.state` value, they will be excluded in the query result.

Query

```
SELECT *
FROM Families.address.state
```

Results

```
[
  "WA",
  "NY"
]
```

WHERE clause

The WHERE clause (`WHERE <filter_condition>`) is optional. It specifies the condition(s) that the JSON documents provided by the source must satisfy in order to be included as part of the result. Any JSON document must evaluate the specified conditions to "true" to be considered for the result. The WHERE clause is used by the index layer in order to determine the absolute smallest subset of source documents that can be part of the result.

The following query requests documents that contain a name property whose value is `AndersenFamily`. Any other document that does not have a name property, or where the value does not match `AndersenFamily` is excluded.

Query

```
SELECT f.address
FROM Families f
WHERE f.id = "AndersenFamily"
```

Results

```
[{
  "address": {
    "state": "WA",
    "county": "King",
    "city": "seattle"
  }
}]
```

The previous example showed a simple equality query. DocumentDB SQL also supports a variety of scalar expressions. The most commonly used are binary and unary expressions. Property references from the source JSON object are also valid expressions.

The following binary operators are currently supported and can be used in queries as shown in the following examples:

Arithmetic	+, -, *, /, %
Bitwise	~, &, ^, <<, >>, >>> (zero-fill right shift)
Logical	AND, OR, NOT
Comparison	=, !=, <, >, <=, >=, <>
String	(concatenate)

Let's take a look at some queries using binary operators.

```
SELECT *
FROM Families.children[0] c
WHERE c.grade % 2 = 1      -- matching grades == 5, 1

SELECT *
FROM Families.children[0] c
WHERE c.grade ^ 4 = 1      -- matching grades == 5

SELECT *
FROM Families.children[0] c
WHERE c.grade >= 5         -- matching grades == 5
```

The unary operators +, -, ~ and NOT are also supported, and can be used inside queries as shown in the following example:

```

SELECT *
FROM Families.children[0] c
WHERE NOT(c.grade = 5) -- matching grades == 1

SELECT *
FROM Families.children[0] c
WHERE (-c.grade = -5) -- matching grades == 5

```

In addition to binary and unary operators, property references are also allowed. For example,

`SELECT * FROM Families f WHERE f.isRegistered` returns the JSON document containing the property `isRegistered` where the property's value is equal to the JSON `true` value. Any other values (false, null, Undefined, `<number>`, `<string>`, `<object>`, `<array>`, etc.) leads to the source document being excluded from the result.

Equality and comparison operators

The following table shows the result of equality comparisons in DocumentDB SQL between any two JSON types.

O p	U n d e f i n e d	N u l l	B o o l e a n	N u m b e r	S t r i n g	O b j e c t	A r r a y
U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d
N u l l	U n d e f i n e d	O K	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d
B o o l e a n	U n d e f i n e d	U n d e f i n e d	O K	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d

N u m b e r	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	O K	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d
S t r i n g	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	O K	U n d e f i n e d	U n d e f i n e d
O b j e c t	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	O K	U n d e f i n e d
A r r a y	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	U n d e f i n e d	O K

For other comparison operators such as >, >=, !=, < and <=, the following rules apply:

- Comparison across types results in Undefined.
- Comparison between two objects or two arrays results in Undefined.

If the result of the scalar expression in the filter is Undefined, the corresponding document would not be included in the result, since Undefined doesn't logically equate to "true".

BETWEEN keyword

You can also use the BETWEEN keyword to express queries against ranges of values like in ANSI SQL. BETWEEN can be used against strings or numbers.

For example, this query returns all family documents in which the first child's grade is between 1-5 (both inclusive).

```
SELECT *
FROM Families.children[0] c
WHERE c.grade BETWEEN 1 AND 5
```

Unlike in ANSI-SQL, you can also use the BETWEEN clause in the FROM clause like in the following example.

```
SELECT (c.grade BETWEEN 0 AND 10)
FROM Families.children[0] c
```

For faster query execution times, remember to create an indexing policy that uses a range index type against any numeric properties/paths that are filtered in the BETWEEN clause.

The main difference between using BETWEEN in DocumentDB and ANSI SQL is that you can express range queries against properties of mixed types – for example, you might have "grade" be a number (5) in some documents and strings ("grade4"). In these cases, like in JavaScript, a comparison between two different types results in "undefined", and the document will be skipped.

Logical (AND, OR and NOT) operators

Logical operators operate on Boolean values. The logical truth tables for these operators are shown in the following tables.

OR	TRUE	FALSE	UNDEFINED
True	True	True	True
False	True	False	Undefined
Undefined	True	Undefined	Undefined

AND	TRUE	FALSE	UNDEFINED
True	True	False	Undefined
False	False	False	False
Undefined	Undefined	False	Undefined

NOT	
True	False
False	True
Undefined	Undefined

IN keyword

The IN keyword can be used to check whether a specified value matches any value in a list. For example, this query returns all family documents where the id is one of "WakefieldFamily" or "AndersenFamily".

```
SELECT *
FROM Families
WHERE Families.id IN ('AndersenFamily', 'WakefieldFamily')
```

This example returns all documents where the state is any of the specified values.

```
SELECT *
FROM Families
WHERE Families.address.state IN ("NY", "WA", "CA", "PA", "OH", "OR", "MI", "WI", "MN", "FL")
```

Ternary (?) and Coalesce (??) operators

The Ternary and Coalesce operators can be used to build conditional expressions, similar to popular programming languages like C# and JavaScript.

The Ternary (?) operator can be very handy when constructing new JSON properties on the fly. For example, now you can write queries to classify the class levels into a human readable form like Beginner/Intermediate/Advanced as shown below.

```
SELECT (c.grade < 5)? "elementary": "other" AS gradeLevel
FROM Families.children[0] c
```

You can also nest the calls to the operator like in the query below.

```
SELECT (c.grade < 5)? "elementary": ((c.grade < 9)? "junior": "high") AS gradeLevel
FROM Families.children[0] c
```

As with other query operators, if the referenced properties in the conditional expression are missing in any document, or if the types being compared are different, then those documents will be excluded in the query results.

The Coalesce (??) operator can be used to efficiently check for the presence of a property (a.k.a. is defined) in a document. This is useful when querying against semi-structured or data of mixed types. For example, this query returns the "lastName" if present, or the "surname" if it isn't present.

```
SELECT f.lastName ?? f.surname AS familyName
FROM Families f
```

Quoted property accessor

You can also access properties using the quoted property operator `[]`. For example, `SELECT c.grade` and `SELECT c["grade"]` are equivalent. This syntax is useful when you need to escape a property that contains spaces, special characters, or happens to share the same name as a SQL keyword or reserved word.

```
SELECT f["lastName"]
FROM Families f
WHERE f["id"] = "AndersenFamily"
```

SELECT clause

The SELECT clause (`SELECT <select_list>`) is mandatory and specifies what values will be retrieved from the query, just like in ANSI-SQL. The subset that's been filtered on top of the source documents are passed onto the projection phase, where the specified JSON values are retrieved and a new JSON object is constructed, for each input passed onto it.

The following example shows a typical SELECT query.

Query

```
SELECT f.address
FROM Families f
WHERE f.id = "AndersenFamily"
```

Results

```
[{
  "address": {
    "state": "WA",
    "county": "King",
    "city": "seattle"
  }
}]
```

Nested properties

In the following example, we are projecting two nested properties `f.address.state` and `f.address.city`.

Query

```
SELECT f.address.state, f.address.city
FROM Families f
WHERE f.id = "AndersenFamily"
```

Results

```
[{
  "state": "WA",
  "city": "seattle"
}]
```

Projection also supports JSON expressions as shown in the following example.

Query

```
SELECT { "state": f.address.state, "city": f.address.city, "name": f.id }
FROM Families f
WHERE f.id = "AndersenFamily"
```

Results

```
[{
  "$1": {
    "state": "WA",
    "city": "seattle",
    "name": "AndersenFamily"
  }
}]
```

Let's look at the role of `$1` here. The `SELECT` clause needs to create a JSON object and since no key is provided, we use implicit argument variable names starting with `$1`. For example, this query returns two implicit argument variables, labeled `$1` and `$2`.

Query

```
SELECT { "state": f.address.state, "city": f.address.city },
       { "name": f.id }
FROM Families f
WHERE f.id = "AndersenFamily"
```

Results

```
[{
  "$1": {
    "state": "WA",
    "city": "seattle"
  },
  "$2": {
    "name": "AndersenFamily"
  }
}]
```

Aliasing

Now let's extend the example above with explicit aliasing of values. AS is the keyword used for aliasing. Note that it's optional as shown while projecting the second value as `NameInfo`.

In case a query has two properties with the same name, aliasing must be used to rename one or both of the properties so that they are disambiguated in the projected result.

Query

```
SELECT
  { "state": f.address.state, "city": f.address.city } AS AddressInfo,
  { "name": f.id } NameInfo
FROM Families f
WHERE f.id = "AndersenFamily"
```

Results

```
[{
  "AddressInfo": {
    "state": "WA",
    "city": "seattle"
  },
  "NameInfo": {
    "name": "AndersenFamily"
  }
}]
```

Scalar expressions

In addition to property references, the SELECT clause also supports scalar expressions like constants, arithmetic expressions, logical expressions, etc. For example, here's a simple "Hello World" query.

Query

```
SELECT "Hello World"
```

Results

```
[{
  "$1": "Hello World"
}]
```

Here's a more complex example that uses a scalar expression.

Query

```
SELECT ((2 + 11 % 7)-2)/3
```

Results

```
[{
  "$1": 1.33333
}]
```

In the following example, the result of the scalar expression is a Boolean.

Query

```
SELECT f.address.city = f.address.state AS AreFromSameCityState
FROM Families f
```

Results

```
[
  {
    "AreFromSameCityState": false
  },
  {
    "AreFromSameCityState": true
  }
]
```

Object and array creation

Another key feature of DocumentDB SQL is array/object creation. In the previous example, note that we created a new JSON object. Similarly, one can also construct arrays as shown in the following examples.

Query

```
SELECT [f.address.city, f.address.state] AS CityState
FROM Families f
```

Results

```
[
  {
    "CityState": [
      "seattle",
      "WA"
    ]
  },
  {
    "CityState": [
      "NY",
      "NY"
    ]
  }
]
```

VALUE keyword

The **VALUE** keyword provides a way to return JSON value. For example, the query shown below returns the scalar `"Hello World"` instead of `{"$1": "Hello World"}`.

Query

```
SELECT VALUE "Hello World"
```

Results

```
[
  "Hello World"
]
```

The following query returns the JSON value without the "address" label in the results.

Query

```
SELECT VALUE f.address
FROM Families f
```

Results

```
[
  {
    "state": "WA",
    "county": "King",
    "city": "seattle"
  },
  {
    "state": "NY",
    "county": "Manhattan",
    "city": "NY"
  }
]
```

The following example extends this to show how to return JSON primitive values (the leaf level of the JSON tree).

Query

```
SELECT VALUE f.address.state
FROM Families f
```

Results

```
[
  "WA",
  "NY"
]
```

* Operator

The special operator (*) is supported to project the document as-is. When used, it must be the only projected field. While a query like `SELECT * FROM Families f` is valid, `SELECT VALUE * FROM Families f` and `SELECT *, f.id FROM Families f` are not valid.

Query

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

Results

```
[{
  "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
}]
```

TOP Operator

The TOP keyword can be used to limit the number of values from a query. When TOP is used in conjunction with the ORDER BY clause, the result set is limited to the first N number of ordered values; otherwise, it returns the first N number of results in an undefined order. As a best practice, in a SELECT statement, always use an ORDER BY clause with the TOP clause. This is the only way to predictably indicate which rows are affected by TOP.

Query

```
SELECT TOP 1 *
FROM Families f
```

Results

```
[{
  "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
}]
```

TOP can be used with a constant value (as shown above) or with a variable value using parameterized queries. For more details, please see parameterized queries below.

ORDER BY clause

Like in ANSI-SQL, you can include an optional Order By clause while querying. The clause can include an optional ASC/DESC argument to specify the order in which results must be retrieved. For a more detailed look at Order By, refer to [DocumentDB Order By Walkthrough](#).

For example, here's a query that retrieves families in order of the resident city's name.

Query

```
SELECT f.id, f.address.city
FROM Families f
ORDER BY f.address.city
```

Results

```
[
  {
    "id": "WakefieldFamily",
    "city": "NY"
  },
  {
    "id": "AndersenFamily",
    "city": "Seattle"
  }
]
```

And here's a query that retrieves families in order of creation date, which is stored as a number representing the epoch time, i.e, elapsed time since Jan 1, 1970 in seconds.

Query

```
SELECT f.id, f.creationDate
FROM Families f
ORDER BY f.creationDate DESC
```

Results

```
[
  {
    "id": "WakefieldFamily",
    "creationDate": 1431620462
  },
  {
    "id": "AndersenFamily",
    "creationDate": 1431620472
  }
]
```

Advanced database concepts and SQL queries

Iteration

A new construct was added via the **IN** keyword in DocumentDB SQL to provide support for iterating over JSON arrays. The FROM source provides support for iteration. Let's start with the following example:

Query

```
SELECT *
FROM Families.children
```

Results

```
[
  [
    {
      "firstName": "Henriette Thaulow",
      "gender": "female",
      "grade": 5,
      "pets": [{ "givenName": "Fluffy"}]
    }
  ],
  [
    {
      "familyName": "Merriam",
      "givenName": "Jesse",
      "gender": "female",
      "grade": 1
    },
    {
      "familyName": "Miller",
      "givenName": "Lisa",
      "gender": "female",
      "grade": 8
    }
  ]
]
```

Now let's look at another query that performs iteration over children in the collection. Note the difference in the output array. This example splits `children` and flattens the results into a single array.

Query

```
SELECT *
FROM c IN Families.children
```

Results

```
[
  {
    "firstName": "Henriette Thaulow",
    "gender": "female",
    "grade": 5,
    "pets": [{ "givenName": "Fluffy" }]
  },
  {
    "familyName": "Merriam",
    "givenName": "Jesse",
    "gender": "female",
    "grade": 1
  },
  {
    "familyName": "Miller",
    "givenName": "Lisa",
    "gender": "female",
    "grade": 8
  }
]
```

This can be further used to filter on each individual entry of the array as shown in the following example.

Query

```
SELECT c.givenName
FROM c IN Families.children
WHERE c.grade = 8
```

Results

```
[{
  "givenName": "Lisa"
}]
```

Joins

In a relational database, the need to join across tables is very important. It's the logical corollary to designing normalized schemas. Contrary to this, DocumentDB deals with the denormalized data model of schema-free documents. This is the logical equivalent of a "self-join".

The syntax that the language supports is `JOIN JOIN ... JOIN`. Overall, this returns a set of **N**-tuples (tuple with **N** values). Each tuple has values produced by iterating all collection aliases over their respective sets. In other words, this is a full cross product of the sets participating in the join.

The following examples show how the `JOIN` clause works. In the following example, the result is empty since the cross product of each document from source and an empty set is empty.

Query

```
SELECT f.id
FROM Families f
JOIN f.NonExistent
```

Results

```
[{
}]
```

In the following example, the join is between the document root and the `children` sub-root. It's a cross product between two JSON objects. The fact that `children` is an array is not effective in the `JOIN` since we are dealing with a single root that is the `children` array. Hence the result contains only two results, since the cross product of each document with the array yields exactly only one document.

Query

```
SELECT f.id
FROM Families f
JOIN f.children
```

Results

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

The following example shows a more conventional join:

Query

```
SELECT f.id
FROM Families f
JOIN c IN f.children
```

Results

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

The first thing to note is that the `from_source` of the **JOIN** clause is an iterator. So, the flow in this case is as follows:

- Expand each child element `c` in the array.
- Apply a cross product with the root of the document `f` with each child element `c` that was flattened in the first step.
- Finally, project the root object `f` name property alone.

The first document (`AndersenFamily`) contains only one child element, so the result set contains only a single object corresponding to this document. The second document (`WakefieldFamily`) contains two children. So, the cross product produces a separate object for each child, thereby resulting in two objects, one for each child corresponding to this document. Note that the root fields in both these documents will be same, just as you would expect in a cross product.

The real utility of the JOIN is to form tuples from the cross-product in a shape that's otherwise difficult to project. Furthermore, as we will see in the example below, you could filter on the combination of a tuple that lets' the user chose a condition satisfied by the tuples overall.

Query

```
SELECT
  f.id AS familyName,
  c.givenName AS childGivenName,
  c.firstName AS childFirstName,
  p.givenName AS petName
FROM Families f
JOIN c IN f.children
JOIN p IN c.pets
```

Results

```
[
  {
    "familyName": "AndersenFamily",
    "childFirstName": "Henriette Thaulow",
    "petName": "Fluffy"
  },
  {
    "familyName": "WakefieldFamily",
    "childGivenName": "Jesse",
    "petName": "Goofy"
  },
  {
    "familyName": "WakefieldFamily",
    "childGivenName": "Jesse",
    "petName": "Shadow"
  }
]
```

This example is a natural extension of the preceding example, and performs a double join. So, the cross product can be viewed as the following pseudo-code.

```
for-each(Family f in Families)
{
  for-each(Child c in f.children)
  {
    for-each(Pet p in c.pets)
    {
      return (Tuple(f.id AS familyName,
                    c.givenName AS childGivenName,
                    c.firstName AS childFirstName,
                    p.givenName AS petName));
    }
  }
}
```

`AndersenFamily` has one child who has one pet. So, the cross product yields one row (1 11) from this family. WakefieldFamily however has two children, but only one child "Jesse" has pets. Jesse has 2 pets though. Hence the cross product yields $1 \times 2 = 2$ rows from this family.

In the next example, there is an additional filter on `pet`. This excludes all the tuples where the pet name is not "Shadow". Notice that we are able to build tuples from arrays, filter on any of the elements of the tuple, and project any combination of the elements.

Query

```
SELECT
  f.id AS familyName,
  c.givenName AS childGivenName,
  c.firstName AS childFirstName,
  p.givenName AS petName
FROM Families f
JOIN c IN f.children
JOIN p IN c.pets
WHERE p.givenName = "Shadow"
```

Results

```
[
  {
    "familyName": "WakefieldFamily",
    "childGivenName": "Jesse",
    "petName": "Shadow"
  }
]
```

JavaScript integration

DocumentDB provides a programming model for executing JavaScript based application logic directly on the collections in terms of stored procedures and triggers. This allows for both:

- Ability to do high performance transactional CRUD operations and queries against documents in a collection by virtue of the deep integration of JavaScript runtime directly within the database engine.
- A natural modeling of control flow, variable scoping, and assignment and integration of exception handling primitives with database transactions. For more details about DocumentDB support for JavaScript integration, please refer to the JavaScript server side programmability documentation.

User Defined Functions (UDFs)

Along with the types already defined in this article, DocumentDB SQL provides support for User Defined Functions (UDF). In particular, scalar UDFs are supported where the developers can pass in zero or many arguments and return a single argument result back. Each of these arguments are checked for being legal JSON values.

The DocumentDB SQL syntax is extended to support custom application logic using these User Defined Functions. UDFs can be registered with DocumentDB and then be referenced as part of a SQL query. In fact, the UDFs are exquisitely designed to be invoked by queries. As a corollary to this choice, UDFs do not have access to the context object which the other JavaScript types (stored procedures and triggers) have. Since queries execute as read-only, they can run either on primary or on secondary replicas. Therefore, UDFs are designed to run on secondary replicas unlike other JavaScript types.

Below is an example of how a UDF can be registered at the DocumentDB database, specifically under a document collection.

```
UserDefinedFunction regexMatchUdf = new UserDefinedFunction
{
    Id = "REGEX_MATCH",
    Body = @"function (input, pattern) {
        return input.match(pattern) != null;
    };",
};

UserDefinedFunction createdUdf = client.CreateUserDefinedFunctionAsync(
    UriFactory.CreateDocumentCollectionUri("testdb", "families"),
    regexMatchUdf).Result;
```

The preceding example creates a UDF whose name is `REGEX_MATCH`. It accepts two JSON string values `input` and `pattern` and checks if the first matches the pattern specified in the second using JavaScript's `string.match()` function.

We can now use this UDF in a query in a projection. UDFs must be qualified with the case-sensitive prefix `"udf."` when called from within queries.

NOTE

Prior to 3/17/2015, DocumentDB supported UDF calls without the "udf." prefix like `SELECT REGEX_MATCH()`. This calling pattern has been deprecated.

Query

```
SELECT udf.REGEX_MATCH(Families.address.city, ".*eattle")
FROM Families
```

Results

```
[
  {
    "$1": true
  },
  {
    "$1": false
  }
]
```

The UDF can also be used inside a filter as shown in the example below, also qualified with the "udf." prefix :

Query

```
SELECT Families.id, Families.address.city
FROM Families
WHERE udf.REGEX_MATCH(Families.address.city, ".*eattle")
```

Results

```
[{
  "id": "AndersenFamily",
  "city": "Seattle"
}]
```

In essence, UDFs are valid scalar expressions and can be used in both projections and filters.

To expand on the power of UDFs, let's look at another example with conditional logic:

```
UserDefinedFunction seaLevelUdf = new UserDefinedFunction()
{
    Id = "SEALEVEL",
    Body = @"function(city) {
        switch (city) {
            case 'seattle':
                return 520;
            case 'NY':
                return 410;
            case 'Chicago':
                return 673;
            default:
                return -1;
        }
    }";
};

UserDefinedFunction createdUdf = await client.CreateUserDefinedFunctionAsync(
    UriFactory.CreateDocumentCollectionUri("testdb", "families"),
    seaLevelUdf);
```

Below is an example that exercises the UDF.

Query

```
SELECT f.address.city, udf.SEALEVEL(f.address.city) AS seaLevel
FROM Families f
```

Results

```
[
  {
    "city": "seattle",
    "seaLevel": 520
  },
  {
    "city": "NY",
    "seaLevel": 410
  }
]
```

As the preceding examples showcase, UDFs integrate the power of JavaScript language with the DocumentDB SQL to provide a rich programmable interface to do complex procedural, conditional logic with the help of inbuilt JavaScript runtime capabilities.

DocumentDB SQL provides the arguments to the UDFs for each document in the source at the current stage (WHERE clause or SELECT clause) of processing the UDF. The result is incorporated in the overall execution pipeline seamlessly. If the properties referred to by the UDF parameters are not available in the JSON value, the parameter is considered as undefined and hence the UDF invocation is entirely skipped. Similarly if the result of the UDF is undefined, it's not included in the result.

In summary, UDFs are great tools to do complex business logic as part of the query.

Operator evaluation

DocumentDB, by the virtue of being a JSON database, draws parallels with JavaScript operators and its evaluation semantics. While DocumentDB tries to preserve JavaScript semantics in terms of JSON support, the operation evaluation deviates in some instances.

In DocumentDB SQL, unlike in traditional SQL, the types of values are often not known until the values are actually retrieved from database. In order to efficiently execute queries, most of the operators have strict type requirements.

DocumentDB SQL doesn't perform implicit conversions, unlike JavaScript. For instance, a query like

```
SELECT * FROM Person p WHERE p.Age = 21
```

 matches documents which contain an Age property whose value is 21. Any other document whose Age property matches string "21", or other possibly infinite variations like "021", "21.0", "0021", "00021", etc. will not be matched. This is in contrast to the JavaScript where the string values are implicitly casted to numbers (based on operator, ex: ==). This choice is crucial for efficient index matching in DocumentDB SQL.

Parameterized SQL queries

DocumentDB supports queries with parameters expressed with the familiar @ notation. Parameterized SQL provides robust handling and escaping of user input, preventing accidental exposure of data through SQL injection.

For example, you can write a query that takes last name and address state as parameters, and then execute it for various values of last name and address state based on user input.

```
SELECT *
FROM Families f
WHERE f.lastName = @lastName AND f.address.state = @addressState
```

This request can then be sent to DocumentDB as a parameterized JSON query like shown below.

```
{
  "query": "SELECT * FROM Families f WHERE f.lastName = @lastName AND f.address.state = @addressState",
  "parameters": [
    {"name": "@lastName", "value": "Wakefield"},
    {"name": "@addressState", "value": "NY"},
  ]
}
```

The argument to TOP can be set using parameterized queries like shown below.

```
{
  "query": "SELECT TOP @n * FROM Families",
  "parameters": [
    {"name": "@n", "value": 10},
  ]
}
```

Parameter values can be any valid JSON (strings, numbers, Booleans, null, even arrays or nested JSON). Also since DocumentDB is schema-less, parameters are not validated against any type.

Built-in functions

DocumentDB also supports a number of built-in functions for common operations, that can be used inside queries like user defined functions (UDFs).

FUNCTION GROUP	OPERATIONS
Mathematical functions	ABS, CEILING, EXP, FLOOR, LOG, LOG10, POWER, ROUND, SIGN, SQRT, SQUARE, TRUNC, ACOS, ASIN, ATAN, ATN2, COS, COT, DEGREES, PI, RADIANS, SIN, and TAN
Type checking functions	IS_ARRAY, IS_BOOL, IS_NULL, IS_NUMBER, IS_OBJECT, IS_STRING, IS_DEFINED, and IS_PRIMITIVE
String functions	CONCAT, CONTAINS, ENDSWITH, INDEX_OF, LEFT, LENGTH, LOWER, LTRIM, REPLACE, REPLICATE, REVERSE, RIGHT, RTRIM, STARTSWITH, SUBSTRING, and UPPER
Array functions	ARRAY_CONCAT, ARRAY_CONTAINS, ARRAY_LENGTH, and ARRAY_SLICE
Spatial functions	ST_DISTANCE, ST_WITHIN, ST_INTERSECTS, ST_ISVALID, and ST_ISVALIDDETAILED

If you're currently using a user defined function (UDF) for which a built-in function is now available, you should use the corresponding built-in function as it is going to be quicker to run and more efficiently.

Mathematical functions

The mathematical functions each perform a calculation, usually based on input values that are provided as arguments, and return a numeric value. Here's a table of supported built-in mathematical functions.

USAGE	DESCRIPTION
<code>[ABS (num_expr)</code>	Returns the absolute (positive) value of the specified numeric expression.
<code>CEILING (num_expr)</code>	Returns the smallest integer value greater than, or equal to, the specified numeric expression.
<code>FLOOR (num_expr)</code>	Returns the largest integer less than or equal to the specified numeric expression.
<code>EXP (num_expr)</code>	Returns the exponent of the specified numeric expression.
<code>LOG (num_expr [,base])</code>	Returns the natural logarithm of the specified numeric expression, or the logarithm using the specified base
<code>LOG10 (num_expr)</code>	Returns the base-10 logarithmic value of the specified numeric expression.
<code>ROUND (num_expr)</code>	Returns a numeric value, rounded to the closest integer value.
<code>TRUNC (num_expr)</code>	Returns a numeric value, truncated to the closest integer value.
<code>SQRT (num_expr)</code>	Returns the square root of the specified numeric expression.
<code>SQUARE (num_expr)</code>	Returns the square of the specified numeric expression.
<code>POWER (num_expr, num_expr)</code>	Returns the power of the specified numeric expression to the value specified.
<code>SIGN (num_expr)</code>	Returns the sign value (-1, 0, 1) of the specified numeric expression.
<code>ACOS (num_expr)</code>	Returns the angle, in radians, whose cosine is the specified numeric expression; also called arccosine.
<code>ASIN (num_expr)</code>	Returns the angle, in radians, whose sine is the specified numeric expression. This is also called arcsine.
<code>ATAN (num_expr)</code>	Returns the angle, in radians, whose tangent is the specified numeric expression. This is also called arctangent.
<code>ATN2 (num_expr)</code>	Returns the angle, in radians, between the positive x-axis and the ray from the origin to the point (y, x), where x and y are the values of the two specified float expressions.
<code>COS (num_expr)</code>	Returns the trigonometric cosine of the specified angle, in radians, in the specified expression.
<code>COT (num_expr)</code>	Returns the trigonometric cotangent of the specified angle, in radians, in the specified numeric expression.

USAGE	DESCRIPTION
<code>DEGREES (num_expr)</code>	Returns the corresponding angle in degrees for an angle specified in radians.
<code>PI ()</code>	Returns the constant value of PI.
<code>RADIANS (num_expr)</code>	Returns radians when a numeric expression, in degrees, is entered.
<code>SIN (num_expr)</code>	Returns the trigonometric sine of the specified angle, in radians, in the specified expression.
<code>TAN (num_expr)</code>	Returns the tangent of the input expression, in the specified expression.

For example, you can now run queries like the following:

Query

```
SELECT VALUE ABS(-4)
```

Results

```
[4]
```

The main difference between DocumentDB's functions compared to ANSI SQL is that they are designed to work well with schema-less and mixed schema data. For example, if you have a document where the Size property is missing, or has a non-numeric value like "unknown", then the document is skipped over, instead of returning an error.

Type checking functions

The type checking functions allow you to check the type of an expression within SQL queries. Type checking functions can be used to determine the type of properties within documents on the fly when it is variable or unknown. Here's a table of supported built-in type checking functions.

Usage	Description
<code>IS_ARRAY (expr)</code>	Returns a Boolean indicating if the type of the value is an array.
<code>IS_BOOL (expr)</code>	Returns a Boolean indicating if the type of the value is a Boolean.
<code>IS_NULL (expr)</code>	Returns a Boolean indicating if the type of the value is null.
<code>IS_NUMBER (expr)</code>	Returns a Boolean indicating if the type of the value is a number.
<code>IS_OBJECT (expr)</code>	Returns a Boolean indicating if the type of the value is a JSON object.
<code>IS_STRING (expr)</code>	Returns a Boolean indicating if the type of the value is a string.

IS_DEFINED (expr)	Returns a Boolean indicating if the property has been assigned a value.
IS_PRIMITIVE (expr)	Returns a Boolean indicating if the type of the value is a string, number, Boolean or null.

Using these functions, you can now run queries like the following:

Query

SELECT VALUE IS_NUMBER(-4)

Results

[true]

String functions

The following scalar functions perform an operation on a string input value and return a string, numeric or Boolean value. Here's a table of built-in string functions:

USAGE	DESCRIPTION
LENGTH (str_expr)	Returns the number of characters of the specified string expression
CONCAT (str_expr, str_expr [, str_expr])	Returns a string that is the result of concatenating two or more string values.
SUBSTRING (str_expr, num_expr, num_expr)	Returns part of a string expression.
STARTSWITH (str_expr, str_expr)	Returns a Boolean indicating whether the first string expression ends with the second
ENDSWITH (str_expr, str_expr)	Returns a Boolean indicating whether the first string expression ends with the second
CONTAINS (str_expr, str_expr)	Returns a Boolean indicating whether the first string expression contains the second.
INDEX_OF (str_expr, str_expr)	Returns the starting position of the first occurrence of the second string expression within the first specified string expression, or -1 if the string is not found.
LEFT (str_expr, num_expr)	Returns the left part of a string with the specified number of characters.
RIGHT (str_expr, num_expr)	Returns the right part of a string with the specified number of characters.
LTRIM (str_expr)	Returns a string expression after it removes leading blanks.
RTRIM (str_expr)	Returns a string expression after truncating all trailing blanks.

USAGE	DESCRIPTION
<code>LOWER (str_expr)</code>	Returns a string expression after converting uppercase character data to lowercase.
<code>UPPER (str_expr)</code>	Returns a string expression after converting lowercase character data to uppercase.
<code>REPLACE (str_expr, str_expr, str_expr)</code>	Replaces all occurrences of a specified string value with another string value.
<code>REPLICATE (str_expr, num_expr)</code>	Repeats a string value a specified number of times.
<code>REVERSE (str_expr)</code>	Returns the reverse order of a string value.

Using these functions, you can now run queries like the following. For example, you can return the family name in uppercase as follows:

Query

```
SELECT VALUE UPPER(Families.id)
FROM Families
```

Results

```
[
  "WAKEFIELDFAMILY",
  "ANDERSENFAMILY"
]
```

Or concatenate strings like in this example:

Query

```
SELECT Families.id, CONCAT(Families.address.city, ",", Families.address.state) AS location
FROM Families
```

Results

```
[{
  "id": "WakefieldFamily",
  "location": "NY,NY"
},
{
  "id": "AndersenFamily",
  "location": "seattle,WA"
}]
```

String functions can also be used in the WHERE clause to filter results, like in the following example:

Query

```
SELECT Families.id, Families.address.city
FROM Families
WHERE STARTSWITH(Families.id, "Wakefield")
```

Results

```
[{
  "id": "WakefieldFamily",
  "city": "NY"
}]
```

Array functions

The following scalar functions perform an operation on an array input value and return numeric, Boolean or array value. Here's a table of built-in array functions:

USAGE	DESCRIPTION
ARRAY_LENGTH (arr_expr)	Returns the number of elements of the specified array expression.
ARRAY_CONCAT (arr_expr, arr_expr [, arr_expr])	Returns an array that is the result of concatenating two or more array values.
ARRAY_CONTAINS (arr_expr, expr)	Returns a Boolean indicating whether the array contains the specified value.
ARRAY_SLICE (arr_expr, num_expr [, num_expr])	Returns part of an array expression.

Array functions can be used to manipulate arrays within JSON. For example, here's a query that returns all documents where one of the parents is "Robin Wakefield".

Query

```
SELECT Families.id
FROM Families
WHERE ARRAY_CONTAINS(Families.parents, { givenName: "Robin", familyName: "Wakefield" })
```

Results

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

Here's another example that uses `ARRAY_LENGTH` to get the number of children per family.

Query

```
SELECT Families.id, ARRAY_LENGTH(Families.children) AS numberOfChildren
FROM Families
```

Results

```
[{
  "id": "WakefieldFamily",
  "numberOfChildren": 2
},
{
  "id": "AndersenFamily",
  "numberOfChildren": 1
}]
```

Spatial functions

DocumentDB supports the following Open Geospatial Consortium (OGC) built-in functions for geospatial querying.

Usage	Description
ST_DISTANCE (point_expr, point_expr)	Returns the distance between the two GeoJSON Point, Polygon, or LineString expressions.
ST_WITHIN (point_expr, polygon_expr)	Returns a Boolean expression indicating whether the first GeoJSON object (Point, Polygon, or LineString) is within the second GeoJSON object (Point, Polygon, or LineString).
ST_INTERSECTS (spatial_expr, spatial_expr)	Returns a Boolean expression indicating whether the two specified GeoJSON objects (Point, Polygon, or LineString) intersect.
ST_ISVALID	Returns a Boolean value indicating whether the specified GeoJSON Point, Polygon, or LineString expression is valid.
ST_ISVALIDDETAILED	Returns a JSON value containing a Boolean value if the specified GeoJSON Point, Polygon, or LineString 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

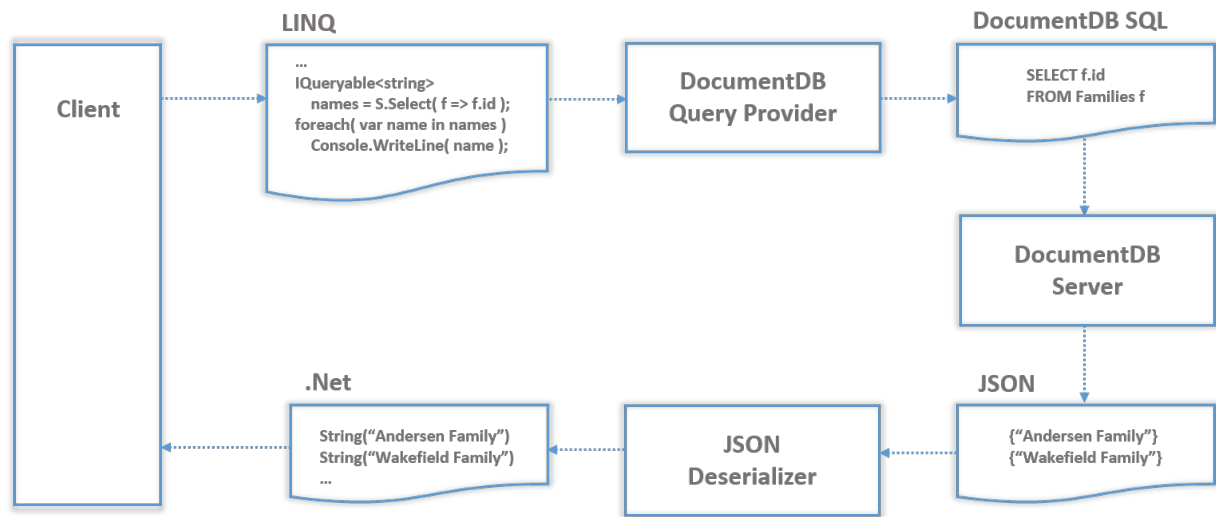
```
[{
  "id": "WakefieldFamily"
}]
```

For more details on geospatial support in DocumentDB, please see [Working with geospatial data in Azure DocumentDB](#). That wraps up spatial functions, and the SQL syntax for DocumentDB. Now let's take a look at how LINQ querying works and how it interacts with the syntax we've seen so far.

LINQ to DocumentDB SQL

LINQ is a .NET programming model that expresses computation as queries on streams of objects. DocumentDB provides a client side library to interface with LINQ by facilitating a conversion between JSON and .NET objects and a mapping from a subset of LINQ queries to DocumentDB queries.

The picture below shows the architecture of supporting LINQ queries using DocumentDB. Using the DocumentDB client, developers can create an **IQueryable** object that directly queries the DocumentDB query provider, which then translates the LINQ query into a DocumentDB query. The query is then passed to the DocumentDB server to retrieve a set of results in JSON format. The returned results are deserialized into a stream of .NET objects on the client side.



.NET and JSON mapping

The mapping between .NET objects and JSON documents is natural - each data member field is mapped to a JSON object, where the field name is mapped to the "key" part of the object and the "value" part is recursively mapped to the value part of the object. Consider the following example. The Family object created is mapped to the JSON document as shown below. And vice versa, the JSON document is mapped back to a .NET object.

C# Class

```

public class Family
{
    [JsonProperty(PropertyName="id")]
    public string Id;
    public Parent[] parents;
    public Child[] children;
    public bool isRegistered;
};

public struct Parent
{
    public string familyName;
    public string givenName;
};

public class Child
{
    public string familyName;
    public string givenName;
    public string gender;
    public int grade;
    public List<Pet> pets;
};

public class Pet
{
    public string givenName;
};

public class Address
{
    public string state;
    public string county;
    public string city;
};

// Create a Family object.
Parent mother = new Parent { familyName= "Wakefield", givenName="Robin" };
Parent father = new Parent { familyName = "Miller", givenName = "Ben" };
Child child = new Child { familyName="Merriam", givenName="Jesse", gender="female", grade=1 };
Pet pet = new Pet { givenName = "Fluffy" };
Address address = new Address { state = "NY", county = "Manhattan", city = "NY" };
Family family = new Family { Id = "WakefieldFamily", parents = new Parent [] { mother, father}, children =
new Child[] { child }, isRegistered = false };

```

JSON

```
{
  "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" },
  "isRegistered": false
};
```

LINQ to SQL translation

The DocumentDB query provider performs a best effort mapping from a LINQ query into a DocumentDB SQL query. In the following description, we assume the reader has a basic familiarity of LINQ.

First, for the type system, we support all JSON primitive types – numeric types, boolean, string, and null. Only these JSON types are supported. The following scalar expressions are supported.

- Constant values – these includes constant values of the primitive data types at the time the query is evaluated.
- Property/array index expressions – these expressions refer to the property of an object or an array element.

family.Id; family.children[0].familyName; family.children[0].grade; family.children[n].grade; //n is an int variable

- Arithmetic expressions - These include common arithmetic expressions on numerical and boolean values. For the complete list, refer to the SQL specification.

2 * family.children[0].grade; x + y;

- String comparison expression - these include comparing a string value to some constant string value.

mother.familyName == "Smith"; child.givenName == s; //s is a string variable

- Object/array creation expression - these expressions return an object of compound value type or anonymous type or an array of such objects. These values can be nested.

new Parent { familyName = "Smith", givenName = "Joe" }; new { first = 1, second = 2 }; //an anonymous type with 2 fields
new int[] { 3, child.grade, 5 };

List of supported LINQ operators

Here is a list of supported LINQ operators in the LINQ provider included with the DocumentDB .NET SDK.

- **Select:** Projections translate to the SQL SELECT including object construction
- **Where:** Filters translate to the SQL WHERE, and support translation between && , || and ! to the SQL operators
- **SelectMany:** Allows unwinding of arrays to the SQL JOIN clause. Can be used to chain/nest expressions to filter on array elements
- **OrderBy and OrderByDescending:** Translates to ORDER BY ascending/descending:
- **CompareTo:** Translates to range comparisons. Commonly used for strings since they're not comparable in .NET
- **Take:** Translates to the SQL TOP for limiting results from a query
- **Math Functions:** Supports translation from .NET's Abs, Acos, Asin, Atan, Ceiling, Cos, Exp, Floor, Log, Log10, Pow, Round, Sign, Sin, Sqrt, Tan, Truncate to the equivalent SQL built-in functions.
- **String Functions:** Supports translation from .NET's Concat, Contains, EndsWith, IndexOf, Count, ToLower, TrimStart, Replace, Reverse, TrimEnd, StartsWith, SubString, ToUpper to the equivalent SQL built-in functions.
- **Array Functions:** Supports translation from .NET's Concat, Contains, and Count to the equivalent SQL built-in functions.
- **Geospatial Extension Functions:** Supports translation from stub methods Distance, Within, IsValid, and IsValidDetailed to the equivalent SQL built-in functions.
- **User Defined Function Extension Function:** Supports translation from the stub method UserDefinedFunctionProvider.Invoke to the corresponding user defined function.
- **Miscellaneous:** Supports translation of the coalesce and conditional operators. Can translate Contains to String CONTAINS, ARRAY_CONTAINS or the SQL IN depending on context.

SQL query operators

Here are some examples that illustrate how some of the standard LINQ query operators are translated down to DocumentDB queries.

Select Operator

The syntax is `input.Select(x => f(x))`, where `f` is a scalar expression.

LINQ lambda expression

```
input.Select(family => family.parents[0].familyName);
```

SQL

```
SELECT VALUE f.parents[0].familyName
FROM Families f
```

LINQ lambda expression

```
input.Select(family => family.children[0].grade + c); // c is an int variable
```

SQL

```
SELECT VALUE f.children[0].grade + c
FROM Families f
```

LINQ lambda expression

```
input.Select(family => new
{
    name = family.children[0].familyName,
    grade = family.children[0].grade + 3
});
```

SQL

```
SELECT VALUE {"name":f.children[0].familyName,
              "grade": f.children[0].grade + 3 }
FROM Families f
```

SelectMany operator

The syntax is `input.SelectMany(x => f(x))`, where `f` is a scalar expression that returns a collection type.

LINQ lambda expression

```
input.SelectMany(family => family.children);
```

SQL

```
SELECT VALUE child
FROM child IN Families.children
```

Where operator

The syntax is `input.Where(x => f(x))`, where `f` is a scalar expression which returns a Boolean value.

LINQ lambda expression

```
input.Where(family=> family.parents[0].familyName == "Smith");
```

SQL

```
SELECT *
FROM Families f
WHERE f.parents[0].familyName = "Smith"
```

LINQ lambda expression

```
input.Where(
    family => family.parents[0].familyName == "Smith" &&
    family.children[0].grade < 3);
```

SQL

```
SELECT *
FROM Families f
WHERE f.parents[0].familyName = "Smith"
AND f.children[0].grade < 3
```

Composite SQL queries

The above operators can be composed to form more powerful queries. Since DocumentDB supports nested collections, the composition can either be concatenated or nested.

Concatenation

The syntax is `input(.|.SelectMany())(.Select())|.Where()*`. A concatenated query can start with an optional

`SelectMany` query followed by multiple `Select` or `Where` operators.

LINQ lambda expression

```
input.Select(family=>family.parents[0])
    .Where(familyName == "Smith");
```

SQL

```
SELECT *
FROM Families f
WHERE f.parents[0].familyName = "Smith"
```

LINQ lambda expression

```
input.Where(family => family.children[0].grade > 3)
    .Select(family => family.parents[0].familyName);
```

SQL

```
SELECT VALUE f.parents[0].familyName
FROM Families f
WHERE f.children[0].grade > 3
```

LINQ lambda expression

```
input.Select(family => new { grade=family.children[0].grade}).
    Where(anon=> anon.grade < 3);
```

SQL

```
SELECT *
FROM Families f
WHERE ({grade: f.children[0].grade}.grade > 3)
```

LINQ lambda expression

```
input.SelectMany(family => family.parents)
    .Where(parent => parents.familyName == "Smith");
```

SQL

```
SELECT *
FROM p IN Families.parents
WHERE p.familyName = "Smith"
```

Nesting

The syntax is `input.SelectMany(x=>x.Q())` where Q is a `Select`, `SelectMany`, or `Where` operator.

In a nested query, the inner query is applied to each element of the outer collection. One important feature is that the inner query can refer to the fields of the elements in the outer collection like self-joins.

LINQ lambda expression

```
input.SelectMany(family=>
    family.parents.Select(p => p.familyName));
```

SQL

```
SELECT VALUE p.familyName
FROM Families f
JOIN p IN f.parents
```

LINQ lambda expression

```
input.SelectMany(family =>
    family.children.Where(child => child.familyName == "Jeff"));
```

SQL

```
SELECT *
FROM Families f
JOIN c IN f.children
WHERE c.familyName = "Jeff"
```

LINQ lambda expression

```
input.SelectMany(family => family.children.Where(
    child => child.familyName == family.parents[0].familyName));
```

SQL

```
SELECT *
FROM Families f
JOIN c IN f.children
WHERE c.familyName = f.parents[0].familyName
```

Executing SQL queries

DocumentDB exposes resources through a REST API that can be called by any language capable of making HTTP/HTTPS requests. Additionally, DocumentDB offers programming libraries for several popular languages like .NET, Node.js, JavaScript and Python. The REST API and the various libraries all support querying through SQL. The .NET SDK supports LINQ querying in addition to SQL.

The following examples show how to create a query and submit it against a DocumentDB database account.

REST API

DocumentDB offers an open RESTful programming model over HTTP. Database accounts can be provisioned using an Azure subscription. The DocumentDB resource model consists of a sets of resources under a database account, each of which is addressable using a logical and stable URI. A set of resources is referred to as a feed in this document. A database account consists of a set of databases, each containing multiple collections, each of which in-turn contain documents, UDFs, and other resource types.

The basic interaction model with these resources is through the HTTP verbs GET, PUT, POST and DELETE with their standard interpretation. The POST verb is used for creation of a new resource, for executing a stored procedure or for issuing a DocumentDB query. Queries are always read only operations with no side-effects.

The following examples show a POST for a DocumentDB query made against a collection containing the two sample documents we've reviewed so far. The query has a simple filter on the JSON name property. Note the use of the `x-ms-documentdb-isquery` and Content-Type: `application/query+json` headers to denote that the operation is a query.

Request

```

POST https://<REST URI>/docs HTTP/1.1
...
x-ms-documentdb-isquery: True
Content-Type: application/query+json

{
  "query": "SELECT * FROM Families f WHERE f.id = @familyId",
  "parameters": [
    { "name": "@familyId", "value": "AndersenFamily" }
  ]
}

```

Results

```

HTTP/1.1 200 Ok
x-ms-activity-id: 8b4678fa-a947-47d3-8dd3-549a40da6eed
x-ms-item-count: 1
x-ms-request-charge: 0.32

<indented for readability, results highlighted>

{
  "_rid": "u1NXANcKogE=",
  "Documents": [
    {
      "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"
      },
      "_rid": "u1NXANcKogEcAAAAAAAAAA==",
      "_ts": 1407691744,
      "_self": "dbs\\/u1NXAA==\\/colls\\/u1NXANcKogE=\\/docs\\/u1NXANcKogEcAAAAAAAAAA==\\/\"",
      "_etag": "\"00002b00-0000-0000-0000-53e7abe00000\"",
      "_attachments": "_attachments\\"
    }
  ],
  "count": 1
}

```

The second example shows a more complex query that returns multiple results from the join.

Request

```

POST https://<REST URI>/docs HTTP/1.1
...
x-ms-documentdb-isquery: True
Content-Type: application/query+json

{
  "query": "SELECT
            f.id AS familyName,
            c.givenName AS childGivenName,
            c.firstName AS childFirstName,
            p.givenName AS petName
          FROM Families f
          JOIN c IN f.children
          JOIN p in c.pets",
  "parameters": []
}

```

Results

```

HTTP/1.1 200 Ok
x-ms-activity-id: 568f34e3-5695-44d3-9b7d-62f8b83e509d
x-ms-item-count: 1
x-ms-request-charge: 7.84

<indented for readability, results highlighted>

{
  "_rid": "u1NXANcKogE=",
  "Documents": [
    {
      "familyName": "AndersenFamily",
      "childFirstName": "Henriette Thaulow",
      "petName": "Fluffy"
    },
    {
      "familyName": "WakefieldFamily",
      "childGivenName": "Jesse",
      "petName": "Goofy"
    },
    {
      "familyName": "WakefieldFamily",
      "childGivenName": "Jesse",
      "petName": "Shadow"
    }
  ],
  "count": 3
}

```

If a query's results cannot fit within a single page of results, then the REST API returns a continuation token through the `x-ms-continuation-token` response header. Clients can paginate results by including the header in subsequent results. The number of results per page can also be controlled through the `x-ms-max-item-count` number header.

To manage the data consistency policy for queries, use the `x-ms-consistency-level` header like all REST API requests. For session consistency, it is required to also echo the latest `x-ms-session-token` Cookie header in the query request. Note that the queried collection's indexing policy can also influence the consistency of query results. With the default indexing policy settings, for collections the index is always current with the document contents and query results will match the consistency chosen for data. If the indexing policy is relaxed to Lazy, then queries can return stale results. For more information, refer to [DocumentDB Consistency Levels](#).

If the configured indexing policy on the collection cannot support the specified query, the DocumentDB server

returns 400 "Bad Request". This is returned for range queries against paths configured for hash (equality) lookups, and for paths explicitly excluded from indexing. The `x-ms-documentdb-query-enable-scan` header can be specified to allow the query to perform a scan when an index is not available.

C# (.NET) SDK

The .NET SDK supports both LINQ and SQL querying. The following example shows how to perform the simple filter query introduced earlier in this document.

```
foreach (var family in client.CreateDocumentQuery(collectionLink,
    "SELECT * FROM Families f WHERE f.id = \"AndersenFamily\""))
{
    Console.WriteLine("\tRead {0} from SQL", family);
}

SqlQuerySpec query = new SqlQuerySpec("SELECT * FROM Families f WHERE f.id = @familyId");
query.Parameters = new SqlParameterCollection();
query.Parameters.Add(new SqlParameter("@familyId", "AndersenFamily"));

foreach (var family in client.CreateDocumentQuery(collectionLink, query))
{
    Console.WriteLine("\tRead {0} from parameterized SQL", family);
}

foreach (var family in (
    from f in client.CreateDocumentQuery(collectionLink)
    where f.Id == "AndersenFamily"
    select f))
{
    Console.WriteLine("\tRead {0} from LINQ query", family);
}

foreach (var family in client.CreateDocumentQuery(collectionLink)
    .Where(f => f.Id == "AndersenFamily")
    .Select(f => f))
{
    Console.WriteLine("\tRead {0} from LINQ lambda", family);
}
```

This sample compares two properties for equality within each document and uses anonymous projections.

```
foreach (var family in client.CreateDocumentQuery(collectionLink,
    @"SELECT {""Name"": f.id, ""City"":f.address.city} AS Family
    FROM Families f
    WHERE f.address.city = f.address.state"))
{
    Console.WriteLine("\tRead {0} from SQL", family);
}

foreach (var family in (
    from f in client.CreateDocumentQuery<Family>(collectionLink)
    where f.address.city == f.address.state
    select new { Name = f.Id, City = f.address.city }))
{
    Console.WriteLine("\tRead {0} from LINQ query", family);
}

foreach (var family in
    client.CreateDocumentQuery<Family>(collectionLink)
    .Where(f => f.address.city == f.address.state)
    .Select(f => new { Name = f.Id, City = f.address.city }))
{
    Console.WriteLine("\tRead {0} from LINQ lambda", family);
}
```

The next sample shows joins, expressed through LINQ SelectMany.

```
foreach (var pet in client.CreateDocumentQuery(collectionLink,
    @"SELECT p
      FROM Families f
      JOIN c IN f.children
      JOIN p in c.pets
      WHERE p.givenName = ""Shadow"""))
{
    Console.WriteLine("\tRead {0} from SQL", pet);
}

// Equivalent in Lambda expressions
foreach (var pet in
    client.CreateDocumentQuery<Family>(collectionLink)
        .SelectMany(f => f.children)
        .SelectMany(c => c.pets)
        .Where(p => p.givenName == "Shadow"))
{
    Console.WriteLine("\tRead {0} from LINQ lambda", pet);
}
```

The .NET client automatically iterates through all the pages of query results in the foreach blocks as shown above. The query options introduced in the REST API section are also available in the .NET SDK using the `FeedOptions` and `FeedResponse` classes in the `CreateDocumentQuery` method. The number of pages can be controlled using the `MaxItemCount` setting.

You can also explicitly control paging by creating `IDocumentQueryable` using the `IQueryable` object, then by reading the `ResponseContinuationToken` values and passing them back as `RequestContinuationToken` in `FeedOptions`. `EnableScanInQuery` can be set to enable scans when the query cannot be supported by the configured indexing policy. For partitioned collections, you can use `PartitionKey` to run the query against a single partition (though DocumentDB can automatically extract this from the query text), and `EnableCrossPartitionQuery` to run queries that may need to be run against multiple partitions.

Refer to [DocumentDB .NET samples](#) for more samples containing queries.

JavaScript server-side API

DocumentDB provides a programming model for executing JavaScript based application logic directly on the collections using stored procedures and triggers. The JavaScript logic registered at a collection level can then issue database operations on the operations on the documents of the given collection. These operations are wrapped in ambient ACID transactions.

The following example show how to use the `queryDocuments` in the JavaScript server API to make queries from inside stored procedures and triggers.

```

function businessLogic(name, author) {
    var context = getContext();
    var collectionManager = context.getCollection();
    var collectionLink = collectionManager.getSelfLink()

    // create a new document.
    collectionManager.createDocument(collectionLink,
        { name: name, author: author },
        function (err, documentCreated) {
            if (err) throw new Error(err.message);

            // filter documents by author
            var filterQuery = "SELECT * from root r WHERE r.author = 'George R.'";
            collectionManager.queryDocuments(collectionLink,
                filterQuery,
                function (err, matchingDocuments) {
                    if (err) throw new Error(err.message);
                    context.getResponse().setBody(matchingDocuments.length);

                    // Replace the author name for all documents that satisfied the query.
                    for (var i = 0; i < matchingDocuments.length; i++) {
                        matchingDocuments[i].author = "George R. R. Martin";
                        // we don't need to execute a callback because they are in parallel
                        collectionManager.replaceDocument(matchingDocuments[i]._self,
                            matchingDocuments[i]);
                    }
                })
        });
}

```

Aggregate functions

Native support for aggregate functions is in the works, but if you need count or sum functionality in the meantime, you can achieve the same result using different methods.

On read path:

- You can perform aggregate functions by retrieving the data and doing a count locally. It's advised to use a cheap query projection like `SELECT VALUE 1` rather than full document such as `SELECT * FROM c`. This helps maximize the number of documents processed in each page of results, thereby avoiding additional round-trips to the service if needed.
- You can also use a stored procedure to minimize network latency on repeated round trips. For a sample stored procedure that calculates the count for a given filter query, see [Count.js](#). The stored procedure can enable users to combine rich business logic along with doing aggregations in an efficient way.

On write path:

- Another common pattern is to pre-aggregate the results in the "write" path. This is especially attractive when the volume of "read" requests is higher than that of "write" requests. Once pre-aggregated, the results are available with a single point read request. The best way to pre-aggregate in DocumentDB is to set up a trigger that is invoked with each "write" and update a metadata document that has the latest results for the query that is being materialized. For instance, please look at the [UpdateaMetadata.js](#) sample, which updates the minSize, maxSize, and totalSize of the metadata document for the collection. The sample can be extended to update a counter, sum, etc.

References

1. [Introduction to Azure DocumentDB](#)
2. [DocumentDB SQL specification](#)

3. [DocumentDB .NET samples](#)
4. [DocumentDB Consistency Levels](#)
5. ANSI SQL 2011 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53681
6. JSON <http://json.org/>
7. Javascript Specification <http://www.ecma-international.org/publications/standards/Ecma-262.htm>
8. LINQ <http://msdn.microsoft.com/library/bb308959.aspx>
9. Query evaluation techniques for large databases <http://dl.acm.org/citation.cfm?id=152611>
10. Query Processing in Parallel Relational Database Systems, IEEE Computer Society Press, 1994
11. Lu, Ooi, Tan, Query Processing in Parallel Relational Database Systems, IEEE Computer Society Press, 1994.
12. Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins: Pig Latin: A Not-So-Foreign Language for Data Processing, SIGMOD 2008.
13. G. Graefe. The Cascades framework for query optimization. IEEE Data Eng. Bull., 18(3): 1995.

DocumentDB server-side programming: Stored procedures, database triggers, and UDFs

11/22/2016 • 25 min to read • [Edit on GitHub](#)

Contributors

[Andrew Liu](#) • [Andy Pasic](#) • [mimig](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [arramac](#) • [v-aljenk](#)

Learn how Azure DocumentDB's language integrated, transactional execution of JavaScript lets developers write **stored procedures, triggers** and **user defined functions (UDFs)** natively in JavaScript. This allows you to write database program application logic that can be shipped and executed directly on the database storage partitions

We recommend getting started by watching the following video, where Andrew Liu provides a brief introduction to DocumentDB's server-side database programming model.

Then, return to this article, where you'll learn the answers to the following questions:

- How do I write a stored procedure, trigger, or UDF using JavaScript?
- How does DocumentDB guarantee ACID?
- How do transactions work in DocumentDB?
- What are pre-triggers and post-triggers and how do I write one?
- How do I register and execute a stored procedure, trigger, or UDF in a RESTful manner by using HTTP?
- What DocumentDB SDKs are available to create and execute stored procedures, triggers, and UDFs?

Introduction to Stored Procedure and UDF Programming

This approach of *"JavaScript as a modern day T-SQL"* frees application developers from the complexities of type system mismatches and object-relational mapping technologies. It also has a number of intrinsic advantages that can be utilized to build rich applications:

- **Procedural Logic:** JavaScript as a high level programming language, provides a rich and familiar interface to express business logic. You can perform complex sequences of operations closer to the data.
- **Atomic Transactions:** DocumentDB guarantees that database operations performed inside a single stored procedure or trigger are atomic. This lets an application combine related operations in a single batch so that either all of them succeed or none of them succeed.
- **Performance:** The fact that JSON is intrinsically mapped to the Javascript language type system and is also the basic unit of storage in DocumentDB allows for a number of optimizations like lazy materialization of JSON documents in the buffer pool and making them available on-demand to the executing code. There are more performance benefits associated with shipping business logic to the database:
 - Batching – Developers can group operations like inserts and submit them in bulk. The network traffic latency cost and the store overhead to create separate transactions are reduced significantly.
 - Pre-compilation – DocumentDB precompiles stored procedures, triggers and user defined functions (UDFs) to avoid JavaScript compilation cost for each invocation. The overhead of building the byte code for the procedural logic is amortized to a minimal value.
 - Sequencing – Many operations need a side-effect (“trigger”) that potentially involves doing one or many secondary store operations. Aside from atomicity, this is more performant when moved to the server.
- **Encapsulation:** Stored procedures can be used to group business logic in one place. This has two advantages:
 - It adds an abstraction layer on top of the raw data, which enables data architects to evolve their applications independently from the data. This is particularly advantageous when the data is schema-less, due to the brittle assumptions that may need to be baked into the application if they have to deal with data directly.
 - This abstraction lets enterprises keep their data secure by streamlining the access from the scripts.

The creation and execution of database triggers, stored procedure and custom query operators is supported through the [REST API](#), [DocumentDB Studio](#), and [client SDKs](#) in many platforms including .NET, Node.js and JavaScript.

This tutorial uses the [Node.js SDK with Q Promises](#) to illustrate syntax and usage of stored procedures, triggers, and UDFs.

Stored procedures

Example: Write a simple stored procedure

Let's start with a simple stored procedure that returns a “Hello World” response.

```
var helloWorldStoredProc = {
  id: "helloWorld",
  body: function () {
    var context = getContext();
    var response = context.getResponse();

    response.setBody("Hello, World");
  }
}
```

Stored procedures are registered per collection, and can operate on any document and attachment present in that collection. The following snippet shows how to register the helloWorld stored procedure with a collection.

```
// register the stored procedure
var createdStoredProcedure;
client.createStoredProcedureAsync('dbs/testdb/colls/testColl', helloWorldStoredProc)
  .then(function (response) {
    createdStoredProcedure = response.resource;
    console.log("Successfully created stored procedure");
  }, function (error) {
    console.log("Error", error);
  });
```

Once the stored procedure is registered, we can execute it against the collection, and read the results back at the client.

```
// execute the stored procedure
client.executeStoredProcedureAsync('dbs/testdb/colls/testColl/sprocs/helloWorld')
  .then(function (response) {
    console.log(response.result); // "Hello, World"
  }, function (err) {
    console.log("Error", error);
  });
```

The context object provides access to all operations that can be performed on DocumentDB storage, as well as access to the request and response objects. In this case, we used the response object to set the body of the response that was sent back to the client. For more details, refer to the [DocumentDB JavaScript server SDK documentation](#).

Let us expand on this example and add more database related functionality to the stored procedure. Stored procedures can create, update, read, query and delete documents and attachments inside the collection.

Example: Write a stored procedure to create a document

The next snippet shows how to use the context object to interact with DocumentDB resources.

```
var createDocumentStoredProc = {
  id: "createMyDocument",
  body: function createMyDocument(documentToCreate) {
    var context = getContext();
    var collection = context.getCollection();

    var accepted = collection.createDocument(collection.getSelfLink(),
      documentToCreate,
      function (err, documentCreated) {
        if (err) throw new Error('Error' + err.message);
        context.getResponse().setBody(documentCreated.id)
      });
    if (!accepted) return;
  }
}
```

This stored procedure takes as input documentToCreate, the body of a document to be created in the current collection. All such operations are asynchronous and depend on JavaScript function callbacks. The callback function has two parameters, one for the error object in case the operation fails, and one for the created object. Inside the callback, users can either handle the exception or throw an error. In case a callback is not provided and there is an error, the DocumentDB runtime throws an error.

In the example above, the callback throws an error if the operation failed. Otherwise, it sets the id of the created document as the body of the response to the client. Here is how this stored procedure is executed with input parameters.

```
// register the stored procedure
client.createStoredProcedureAsync('dbs/testdb/colls/testColl', createDocumentStoredProc)
  .then(function (response) {
    var createdStoredProcedure = response.resource;

    // run stored procedure to create a document
    var docToCreate = {
      id: "DocFromSproc",
      book: "The Hitchhiker's Guide to the Galaxy",
      author: "Douglas Adams"
    };

    return client.executeStoredProcedureAsync('dbs/testdb/colls/testColl/sprocs/createMyDocument',
      docToCreate);
  }, function (error) {
    console.log("Error", error);
  })
  .then(function (response) {
    console.log(response); // "DocFromSproc"
  }, function (error) {
    console.log("Error", error);
  });
```

Note that this stored procedure can be modified to take an array of document bodies as input and create them all in the same stored procedure execution instead of multiple network requests to create each of them individually. This can be used to implement an efficient bulk importer for DocumentDB (discussed later in this tutorial).

The example described demonstrated how to use stored procedures. We will cover triggers and user defined functions (UDFs) later in the tutorial.

Database program transactions

Transaction in a typical database can be defined as a sequence of operations performed as a single logical unit of work. Each transaction provides **ACID guarantees**. ACID is a well-known acronym that stands for four properties - Atomicity, Consistency, Isolation and Durability.

Briefly, atomicity guarantees that all the work done inside a transaction is treated as a single unit where either all of it is committed or none. Consistency makes sure that the data is always in a good internal state across transactions. Isolation guarantees that no two transactions interfere with each other – generally, most commercial systems provide multiple isolation levels that can be used based on the application needs. Durability ensures that any change that's committed in the database will always be present.

In DocumentDB, JavaScript is hosted in the same memory space as the database. Hence, requests made within stored procedures and triggers execute in the same scope of a database session. This enables DocumentDB to guarantee ACID for all operations that are part of a single stored procedure/trigger. Consider the following stored procedure definition:

```

// JavaScript source code
var exchangeItemsSproc = {
  name: "exchangeItems",
  body: function (playerId1, playerId2) {
    var context = getContext();
    var collection = context.getCollection();
    var response = context.getResponse();

    var player1Document, player2Document;

    // query for players
    var filterQuery = 'SELECT * FROM Players p where p.id = \'' + playerId1 + '\'';
    var accept = collection.queryDocuments(collection.getSelfLink(), filterQuery, {},
      function (err, documents, responseOptions) {
        if (err) throw new Error("Error" + err.message);

        if (documents.length != 1) throw "Unable to find both names";
        player1Document = documents[0];

        var filterQuery2 = 'SELECT * FROM Players p where p.id = \'' + playerId2 + '\'';
        var accept2 = collection.queryDocuments(collection.getSelfLink(), filterQuery2, {},
          function (err2, documents2, responseOptions2) {
            if (err2) throw new Error("Error" + err2.message);
            if (documents2.length != 1) throw "Unable to find both names";
            player2Document = documents2[0];
            swapItems(player1Document, player2Document);
            return;
          });
        if (!accept2) throw "Unable to read player details, abort ";
      });

    if (!accept) throw "Unable to read player details, abort ";

    // swap the two players' items
    function swapItems(player1, player2) {
      var player1ItemSave = player1.item;
      player1.item = player2.item;
      player2.item = player1ItemSave;

      var accept = collection.replaceDocument(player1._self, player1,
        function (err, docReplaced) {
          if (err) throw "Unable to update player 1, abort ";

          var accept2 = collection.replaceDocument(player2._self, player2,
            function (err2, docReplaced2) {
              if (err) throw "Unable to update player 2, abort "
            });

          if (!accept2) throw "Unable to update player 2, abort";
        });

      if (!accept) throw "Unable to update player 1, abort";
    }
  }
}

// register the stored procedure in Node.js client
client.createStoredProcedureAsync(collection._self, exchangeItemsSproc)
  .then(function (response) {
    var createdStoredProcedure = response.resource;
  })
);

```

This stored procedure uses transactions within a gaming app to trade items between two players in a single operation. The stored procedure attempts to read two documents each corresponding to the player IDs passed in as an argument. If both player documents are found, then the stored procedure updates the documents by

swapping their items. If any errors are encountered along the way, it throws a JavaScript exception that implicitly aborts the transaction.

If the collection the stored procedure is registered against is a single-partition collection, then the transaction is scoped to all the documents within the collection. If the collection is partitioned, then stored procedures are executed in the transaction scope of a single partition key. Each stored procedure execution must then include a partition key value corresponding to the scope the transaction must run under. For more details, see [DocumentDB Partitioning](#).

Commit and rollback

Transactions are deeply and natively integrated into DocumentDB's JavaScript programming model. Inside a JavaScript function, all operations are automatically wrapped under a single transaction. If the JavaScript completes without any exception, the operations to the database are committed. In effect, the "BEGIN TRANSACTION" and "COMMIT TRANSACTION" statements in relational databases are implicit in DocumentDB.

If there is any exception that's propagated from the script, DocumentDB's JavaScript runtime will roll back the whole transaction. As shown in the earlier example, throwing an exception is effectively equivalent to a "ROLLBACK TRANSACTION" in DocumentDB.

Data consistency

Stored procedures and triggers are always executed on the primary replica of the DocumentDB collection. This ensures that reads from inside stored procedures offer strong consistency. Queries using user defined functions can be executed on the primary or any secondary replica, but we ensure to meet the requested consistency level by choosing the appropriate replica.

Bounded execution

All DocumentDB operations must complete within the server specified request timeout duration. This constraint also applies to JavaScript functions (stored procedures, triggers and user-defined functions). If an operation does not complete with that time limit, the transaction is rolled back. JavaScript functions must finish within the time limit or implement a continuation based model to batch/resume execution.

In order to simplify development of stored procedures and triggers to handle time limits, all functions under the collection object (for create, read, replace, and delete of documents and attachments) return a Boolean value that represents whether that operation will complete. If this value is false, it is an indication that the time limit is about to expire and that the procedure must wrap up execution. Operations queued prior to the first unaccepted store operation are guaranteed to complete if the stored procedure completes in time and does not queue any more requests.

JavaScript functions are also bounded on resource consumption. DocumentDB reserves throughput per collection based on the provisioned size of a database account. Throughput is expressed in terms of a normalized unit of CPU, memory and IO consumption called request units or RUs. JavaScript functions can potentially use up a large number of RUs within a short time, and might get rate-limited if the collection's limit is reached. Resource intensive stored procedures might also be quarantined to ensure availability of primitive database operations.

Example: Bulk importing data into a database program

Below is an example of a stored procedure that is written to bulk-import documents into a collection. Note how the stored procedure handles bounded execution by checking the Boolean return value from `createDocument`, and then uses the count of documents inserted in each invocation of the stored procedure to track and resume progress across batches.

```

function bulkImport(docs) {
  var collection = getContext().getCollection();
  var collectionLink = collection.getSelfLink();

  // The count of imported docs, also used as current doc index.
  var count = 0;

  // Validate input.
  if (!docs) throw new Error("The array is undefined or null.");

  var docsLength = docs.length;
  if (docsLength == 0) {
    getContext().getResponse().setBody(0);
  }

  // Call the create API to create a document.
  tryCreate(docs[count], callback);

  // Note that there are 2 exit conditions:
  // 1) The createDocument request was not accepted.
  //    In this case the callback will not be called, we just call setBody and we are done.
  // 2) The callback was called docs.length times.
  //    In this case all documents were created and we don't need to call tryCreate anymore. Just call
  //    setBody and we are done.
  function tryCreate(doc, callback) {
    var isAccepted = collection.createDocument(collectionLink, doc, callback);

    // If the request was accepted, callback will be called.
    // Otherwise report current count back to the client,
    // which will call the script again with remaining set of docs.
    if (!isAccepted) getContext().getResponse().setBody(count);
  }

  // This is called when collection.createDocument is done in order to process the result.
  function callback(err, doc, options) {
    if (err) throw err;

    // One more document has been inserted, increment the count.
    count++;

    if (count >= docsLength) {
      // If we created all documents, we are done. Just set the response.
      getContext().getResponse().setBody(count);
    } else {
      // Create next document.
      tryCreate(docs[count], callback);
    }
  }
}

```

Database triggers

Database pre-triggers

DocumentDB provides triggers that are executed or triggered by an operation on a document. For example, you can specify a pre-trigger when you are creating a document – this pre-trigger will run before the document is created. The following is an example of how pre-triggers can be used to validate the properties of a document that is being created:

```

var validateDocumentContentsTrigger = {
  name: "validateDocumentContents",
  body: function validate() {
    var context = getContext();
    var request = context.getRequest();

    // document to be created in the current operation
    var documentToCreate = request.getBody();

    // validate properties
    if (!("timestamp" in documentToCreate)) {
      var ts = new Date();
      documentToCreate["my timestamp"] = ts.getTime();
    }

    // update the document that will be created
    request.setBody(documentToCreate);
  },
  triggerType: TriggerType.Pre,
  triggerOperation: TriggerOperation.Create
}

```

And the corresponding Node.js client-side registration code for the trigger:

```

// register pre-trigger
client.createTriggerAsync(collection.self, validateDocumentContentsTrigger)
  .then(function (response) {
    console.log("Created", response.resource);
    var docToCreate = {
      id: "DocWithTrigger",
      event: "Error",
      source: "Network outage"
    };

    // run trigger while creating above document
    var options = { preTriggerInclude: "validateDocumentContents" };

    return client.createDocumentAsync(collection.self,
      docToCreate, options);
  }, function (error) {
    console.log("Error", error);
  })
  .then(function (response) {
    console.log(response.resource); // document with timestamp property added
  }, function (error) {
    console.log("Error", error);
  });

```

Pre-triggers cannot have any input parameters. The request object can be used to manipulate the request message associated with the operation. Here, the pre-trigger is being run with the creation of a document, and the request message body contains the document to be created in JSON format.

When triggers are registered, users can specify the operations that it can run with. This trigger was created with `TriggerOperation.Create`, which means the following is not permitted.

```

var options = { preTriggerInclude: "validateDocumentContents" };

client.replaceDocumentAsync(docToReplace.self,
    newDocBody, options)
    .then(function (response) {
        console.log(response.resource);
    }, function (error) {
        console.log("Error", error);
    });

// Fails, can't use a create trigger in a replace operation

```

Database post-triggers

Post-triggers, like pre-triggers, are associated with an operation on a document and don't take any input parameters. They run **after** the operation has completed, and have access to the response message that is sent to the client.

The following example shows post-triggers in action:

```

var updateMetadataTrigger = {
    name: "updateMetadata",
    body: function updateMetadata() {
        var context = getContext();
        var collection = context.getCollection();
        var response = context.getResponse();

        // document that was created
        var createdDocument = response.getBody();

        // query for metadata document
        var filterQuery = 'SELECT * FROM root r WHERE r.id = "_metadata"';
        var accept = collection.queryDocuments(collection.getSelfLink(), filterQuery,
            updateMetadataCallback);
        if(!accept) throw "Unable to update metadata, abort";

        function updateMetadataCallback(err, documents, responseOptions) {
            if(err) throw new Error("Error" + err.message);
            if(documents.length != 1) throw 'Unable to find metadata document';

            var metadataDocument = documents[0];

            // update metadata
            metadataDocument.createdDocuments += 1;
            metadataDocument.createdNames += " " + createdDocument.id;
            var accept = collection.replaceDocument(metadataDocument._self,
                metadataDocument, function(err, docReplaced) {
                    if(err) throw "Unable to update metadata, abort";
                });
            if(!accept) throw "Unable to update metadata, abort";
            return;
        }
    },
    triggerType: TriggerType.Post,
    triggerOperation: TriggerOperation.All
}

```

The trigger can be registered as shown in the following sample.

```
// register post-trigger
client.createTriggerAsync('dbs/testdb/colls/testColl', updateMetadataTrigger)
  .then(function(createdTrigger) {
    var docToCreate = {
      name: "artist_profile_1023",
      artist: "The Band",
      albums: ["Hellujah", "Rotators", "Spinning Top"]
    };

    // run trigger while creating above document
    var options = { postTriggerInclude: "updateMetadata" };

    return client.createDocumentAsync(collection.self,
      docToCreate, options);
  }, function(error) {
    console.log("Error" , error);
  })
  .then(function(response) {
    console.log(response.resource);
  }, function(error) {
    console.log("Error" , error);
  });
```

This trigger queries for the metadata document and updates it with details about the newly created document.

One thing that is important to note is the **transactional** execution of triggers in DocumentDB. This post-trigger runs as part of the same transaction as the creation of the original document. Therefore, if we throw an exception from the post-trigger (say if we are unable to update the metadata document), the whole transaction will fail and be rolled back. No document will be created, and an exception will be returned.

User-defined functions

User-defined functions (UDFs) are used to extend the DocumentDB SQL query language grammar and implement custom business logic. They can only be called from inside queries. They do not have access to the context object and are meant to be used as compute-only JavaScript. Therefore, UDFs can be run on secondary replicas of the DocumentDB service.

The following sample creates a UDF to calculate income tax based on rates for various income brackets, and then uses it inside a query to find all people who paid more than \$20,000 in taxes.

```
var taxUdf = {
  name: "tax",
  body: function tax(income) {

    if(income == undefined)
      throw 'no input';

    if (income < 1000)
      return income * 0.1;
    else if (income < 10000)
      return income * 0.2;
    else
      return income * 0.4;
  }
}
```

The UDF can subsequently be used in queries like in the following sample:

```
// register UDF
client.createUserDefinedFunctionAsync('dbs/testdb/colls/testColl', taxUdf)
  .then(function(response) {
    console.log("Created", response.resource);

    var query = 'SELECT * FROM TaxPayers t WHERE udf.tax(t.income) > 20000';
    return client.queryDocuments('dbs/testdb/colls/testColl',
      query).toArrayAsync();
  }, function(error) {
    console.log("Error" , error);
  })
  .then(function(response) {
    var documents = response.feed;
    console.log(response.resource);
  }, function(error) {
    console.log("Error" , error);
  });
```

JavaScript language-integrated query API

In addition to issuing queries using DocumentDB's SQL grammar, the server-side SDK allows you to perform optimized queries using a fluent JavaScript interface without any knowledge of SQL. The JavaScript query API allows you to programmatically build queries by passing predicate functions into chainable function calls, with a syntax familiar to ECMAScript5's Array built-ins and popular JavaScript libraries like Lodash. Queries are parsed by the JavaScript runtime to be executed efficiently using DocumentDB's indices.

NOTE

`__` (double-underscore) is an alias to `getContext().getCollection()`. In other words, you can use `__` or `getContext().getCollection()` to access the JavaScript query API.

Supported functions include:

- **chain()value([callback] [, options])**
 - Starts a chained call which must be terminated with value().
- **filter(predicateFunction [, options] [, callback])**
 - Filters the input using a predicate function which returns true/false in order to filter in/out input documents into the resulting set. This behaves similar to a WHERE clause in SQL.
- **map(transformationFunction [, options] [, callback])**
 - Applies a projection given a transformation function which maps each input item to a JavaScript object or value. This behaves similar to a SELECT clause in SQL.
- **pluck([propertyName] [, options] [, callback])**
 - This is a shortcut for a map which extracts the value of a single property from each input item.
- **flatten([isShallow] [, options] [, callback])**
 - Combines and flattens arrays from each input item in to a single array. This behaves similar to SelectMany in LINQ.
- **sortBy([predicate] [, options] [, callback])**
 - Produce a new set of documents by sorting the documents in the input document stream in ascending order using the given predicate. This behaves similar to a ORDER BY clause in SQL.
- **sortByDescending([predicate] [, options] [, callback])**
 - Produce a new set of documents by sorting the documents in the input document stream in descending order using the given predicate. This behaves similar to a ORDER BY x DESC clause in SQL.

When included inside predicate and/or selector functions, the following JavaScript constructs get automatically

optimized to run directly on DocumentDB indices:

- Simple operators: = + - * / % | ^ & == != === !== < > <= >= || && << >> >>>! ~
- Literals, including the object literal: {}
- var, return

The following JavaScript constructs do not get optimized for DocumentDB indices:

- Control flow (e.g. if, for, while)
- Function calls

For more information, please see our [Server-Side JS Docs](#).

Example: Write a stored procedure using the JavaScript query API

The following code sample is an example of how the JavaScript Query API can be used in the context of a stored procedure. The stored procedure inserts a document, given by an input parameter, and updates a metadata document, using the `__.filter()` method, with minSize, maxSize, and totalSize based upon the input document's size property.

```

/**
 * Insert actual doc and update metadata doc: minSize, maxSize, totalSize based on doc.size.
 */
function insertDocumentAndUpdateMetadata(doc) {
  // HTTP error codes sent to our callback function by DocDB server.
  var ErrorCode = {
    RETRY_WITH: 449,
  }

  var isAccepted = __.createDocument(__.getSelfLink(), doc, {}, function(err, doc, options) {
    if (err) throw err;

    // Check the doc (ignore docs with invalid/zero size and metaDoc itself) and call updateMetadata.
    if (!doc.isMetadata && doc.size > 0) {
      // Get the meta document. We keep it in the same collection. it's the only doc that has .isMetadata =
      true.
      var result = __.filter(function(x) {
        return x.isMetadata === true
      }, function(err, feed, options) {
        if (err) throw err;

        // We assume that metadata doc was pre-created and must exist when this script is called.
        if (!feed || !feed.length) throw new Error("Failed to find the metadata document.");

        // The metadata document.
        var metaDoc = feed[0];

        // Update metaDoc.minSize:
        // for 1st document use doc.size, for all the rest see if it's less than last min.
        if (metaDoc.minSize == 0) metaDoc.minSize = doc.size;
        else metaDoc.minSize = Math.min(metaDoc.minSize, doc.size);

        // Update metaDoc.maxSize.
        metaDoc.maxSize = Math.max(metaDoc.maxSize, doc.size);

        // Update metaDoc.totalSize.
        metaDoc.totalSize += doc.size;

        // Update/replace the metadata document in the store.
        var isAccepted = __.replaceDocument(metaDoc._self, metaDoc, function(err) {
          if (err) throw err;
          // Note: in case concurrent updates causes conflict with ErrorCode.RETRY_WITH, we can't read the meta
          again
          // and update again because due to Snapshot isolation we will read same exact version (we are
          in same transaction).
          // We have to take care of that on the client side.
        });
        if (!isAccepted) throw new Error("replaceDocument(metaDoc) returned false.");
      });
      if (!result.isAccepted) throw new Error("filter for metaDoc returned false.");
    }
  });
  if (!isAccepted) throw new Error("createDocument(actual doc) returned false.");
}

```

SQL to Javascript cheat sheet

The following table presents various SQL queries and the corresponding JavaScript queries.

As with SQL queries, document property keys (e.g. `doc.id`) are case-sensitive.

SQL	JAVASCRIPT QUERY API	DESCRIPTION BELOW
SELECT * FROM docs	__.map(function(doc) { return doc; });	1
SELECT docs.id, docs.message AS msg, docs.actions FROM docs	__.map(function(doc) { return { id: doc.id, msg: doc.message, actions: doc.actions }; });	2
SELECT * FROM docs WHERE docs.id="X998_Y998"	__.filter(function(doc) { return doc.id === "X998_Y998"; });	3
SELECT * FROM docs WHERE ARRAY_CONTAINS(docs.Tags, 123)	__.filter(function(x) { return x.Tags && x.Tags.indexOf(123) > -1; });	4
SELECT docs.id, docs.message AS msg FROM docs WHERE docs.id="X998_Y998"	__.chain() .filter(function(doc) { return doc.id === "X998_Y998"; }) .map(function(doc) { return { id: doc.id, msg: doc.message }; }) .value();	5
SELECT VALUE tag FROM docs JOIN tag IN docs.Tags ORDER BY docs._ts	__.chain() .filter(function(doc) { return doc.Tags && Array.isArray(doc.Tags); }) .sortBy(function(doc) { return doc._ts; }) .pluck("Tags") .flatten() .value()	6

The following descriptions explain each query in the table above.

1. Results in all documents (paginated with continuation token) as is.
2. Projects the id, message (aliased to msg), and action from all documents.
3. Queries for documents with the predicate: id = "X998_Y998".
4. Queries for documents that have a Tags property and Tags is an array containing the value 123.
5. Queries for documents with a predicate, id = "X998_Y998", and then projects the id and message (aliased to msg).
6. Filters for documents which have an array property, Tags, and sorts the resulting documents by the _ts timestamp system property, and then projects + flattens the Tags array.

Runtime support

[DocumentDB JavaScript server side SDK](#) provides support for the most of the mainstream JavaScript language features as standardized by [ECMA-262](#).

Security

JavaScript stored procedures and triggers are sandboxed so that the effects of one script do not leak to the other without going through the snapshot transaction isolation at the database level. The runtime environments are pooled but cleaned of the context after each run. Hence they are guaranteed to be safe of any unintended side effects from each other.

Pre-compilation

Stored procedures, triggers and UDFs are implicitly precompiled to the byte code format in order to avoid compilation cost at the time of each script invocation. This ensures invocations of stored procedures are fast and have a low footprint.

Client SDK support

In addition to the [Node.js](#) client, DocumentDB supports [.NET](#), [.NET Core](#), [Java](#), [JavaScript](#), and [Python SDKs](#). Stored procedures, triggers and UDFs can be created and executed using any of these SDKs as well. The following example shows how to create and execute a stored procedure using the .NET client. Note how the .NET types are passed into the stored procedure as JSON and read back.

```
var markAntiquesSproc = new StoredProcedure
{
    Id = "ValidateDocumentAge",
    Body = @"
        function(docToCreate, antiqueYear) {
            var collection = getContext().getCollection();
            var response = getContext().getResponse();

            if(docToCreate.Year != undefined && docToCreate.Year < antiqueYear){
                docToCreate.antique = true;
            }

            collection.createDocument(collection.getSelfLink(), docToCreate, {},
                function(err, docCreated, options) {
                    if(err) throw new Error('Error while creating document: ' + err.message);
                    if(options.maxCollectionSizeInMb == 0) throw 'max collection size not found';
                    response.setBody(docCreated);
                });
        }
    ";
};

// register stored procedure
StoredProcedure createdStoredProcedure = await
client.CreateStoredProcedureAsync(UriFactory.CreateDocumentCollectionUri("db", "coll"), markAntiquesSproc);
dynamic document = new Document() { Id = "Borges_112" };
document.Title = "Aleph";
document.Year = 1949;

// execute stored procedure
Document createdDocument = await client.ExecuteStoredProcedureAsync<Document>
(UriFactory.CreateStoredProcedureUri("db", "coll", "sproc"), document, 1920);
```

This sample shows how to use the [.NET SDK](#) to create a pre-trigger and create a document with the trigger enabled.

```

Trigger preTrigger = new Trigger()
{
    Id = "CapitalizeName",
    Body = @"function() {
        var item = getContext().getRequest().getBody();
        item.id = item.id.toUpperCase();
        getContext().getRequest().setBody(item);
    }",
    TriggerOperation = TriggerOperation.Create,
    TriggerType = TriggerType.Pre
};

Document createdItem = await client.CreateDocumentAsync(UriFactory.CreateDocumentCollectionUri("db", "coll"),
new Document { Id = "documentdb" },
    new RequestOptions
    {
        PreTriggerInclude = new List<string> { "CapitalizeName" },
    });

```

And the following example shows how to create a user defined function (UDF) and use it in a [DocumentDB SQL query](#).

```

UserDefinedFunction function = new UserDefinedFunction()
{
    Id = "LOWER",
    Body = @"function(input)
    {
        return input.toLowerCase();
    }"
};

foreach (Book book in client.CreateDocumentQuery(UriFactory.CreateDocumentCollectionUri("db", "coll"),
    "SELECT * FROM Books b WHERE udf.LOWER(b.Title) = 'war and peace'"))
{
    Console.WriteLine("Read {0} from query", book);
}

```

REST API

All DocumentDB operations can be performed in a RESTful manner. Stored procedures, triggers and user-defined functions can be registered under a collection by using HTTP POST. The following is an example of how to register a stored procedure:

```

POST https://<url>/sprocs/ HTTP/1.1
authorization: <<auth>>
x-ms-date: Thu, 07 Aug 2014 03:43:10 GMT

var x = {
    "name": "createAndAddProperty",
    "body": function (docToCreate, addedPropertyName, addedPropertyValue) {
        var collectionManager = getContext().getCollection();
        collectionManager.createDocument(
            collectionManager.getSelfLink(),
            docToCreate,
            function(err, docCreated) {
                if(err) throw new Error('Error: ' + err.message);
                docCreated[addedPropertyName] = addedPropertyValue;
                getContext().getResponse().setBody(docCreated);
            });
    }
}

```

The stored procedure is registered by executing a POST request against the URI `db/testdb/colls/testColl/sprocs` with the body containing the stored procedure to create. Triggers and UDFs can be registered similarly by issuing a POST against `/triggers` and `/udfs` respectively. This stored procedure can then be executed by issuing a POST request against its resource link:

```
POST https://<url>/sprocs/<sproc> HTTP/1.1
authorization: <<auth>>
x-ms-date: Thu, 07 Aug 2014 03:43:20 GMT

[ { "name": "TestDocument", "book": "Autumn of the Patriarch"}, "Price", 200 ]
```

Here, the input to the stored procedure is passed in the request body. Note that the input is passed as a JSON array of input parameters. The stored procedure takes the first input as a document that is a response body. The response we receive is as follows:

```
HTTP/1.1 200 OK

{
  name: 'TestDocument',
  book: 'Autumn of the Patriarch',
  id: 'V7tQANV3rAkDAAAAAAAAAA==',
  ts: 1407830727,
  self: 'db/V7tQAA==/colls/V7tQANV3rAk=/docs/V7tQANV3rAkDAAAAAAAAAA==/',
  etag: '6c006596-0000-0000-0000-53e9cac70000',
  attachments: 'attachments/',
  Price: 200
}
```

Triggers, unlike stored procedures, cannot be executed directly. Instead they are executed as part of an operation on a document. We can specify the triggers to run with a request using HTTP headers. The following is request to create a document.

```
POST https://<url>/docs/ HTTP/1.1
authorization: <<auth>>
x-ms-date: Thu, 07 Aug 2014 03:43:10 GMT
x-ms-documentdb-pre-trigger-include: validateDocumentContents
x-ms-documentdb-post-trigger-include: bookCreationPostTrigger

{
  "name": "newDocument",
  "title": "The Wizard of Oz",
  "author": "Frank Baum",
  "pages": 92
}
```

Here the pre-trigger to be run with the request is specified in the `x-ms-documentdb-pre-trigger-include` header. Correspondingly, any post-triggers are given in the `x-ms-documentdb-post-trigger-include` header. Note that both pre- and post-triggers can be specified for a given request.

Sample code

You can find more server-side code examples (including [bulk-delete](#), and [update](#)) on our [Github repository](#).

Want to share your awesome stored procedure? Please, send us a pull-request!

Next steps

Once you have one or more stored procedures, triggers, and user-defined functions created, you can load them

and view them in the Azure Portal using Script Explorer. For more information, see [View stored procedures, triggers, and user-defined functions using the DocumentDB Script Explorer](#).

You may also find the following references and resources useful in your path to learn more about DocumentDB server-side programming:

- [Azure DocumentDB SDKs](#)
- [DocumentDB Studio](#)
- [JSON](#)
- [JavaScript ECMA-262](#)
- [JavaScript – JSON type system](#)
- [Secure and Portable Database Extensibility](#)
- [Service Oriented Database Architecture](#)
- [Hosting the .NET Runtime in Microsoft SQL server](#)

Performance and scale testing with Azure DocumentDB

11/15/2016 • 4 min to read • [Edit on GitHub](#)

Contributors

[arramac](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [carolinacmoravia](#) • [mimig](#) • [Andrew Hoh](#)

Performance and scale testing is a key step in application development. For many applications, the database tier has a significant impact on the overall performance and scalability, and is therefore a critical component of performance testing. [Azure DocumentDB](#) is purpose-built for elastic scale and predictable performance, and therefore a great fit for applications that need a high-performance database tier.

This article is a reference for developers implementing performance test suites for their DocumentDB workloads, or evaluating DocumentDB for high-performance application scenarios. It focuses primarily on isolated performance testing of the database, but also includes best practices for production applications.

After reading this article, you will be able to answer the following questions:

- Where can I find a sample .NET client application for performance testing of Azure DocumentDB?
- How do I achieve high throughput levels with Azure DocumentDB from my client application?

To get started with code, please download the project from [DocumentDB Performance Testing Sample](#).

NOTE

The goal of this application is to demonstrate best practices for extracting better performance out of DocumentDB with a small number of client machines. This was not made to demonstrate the peak capacity of the service, which can scale limitlessly.

If you're looking for client-side configuration options to improve DocumentDB performance, see [DocumentDB performance tips](#).

Run the performance testing application

The quickest way to get started is to compile and run the .NET sample below, as described in the steps below. You can also review the source code and implement similar configurations to your own client applications.

Step 1: Download the project from [DocumentDB Performance Testing Sample](#), or fork the Github repository.

Step 2: Modify the settings for `EndpointUrl`, `AuthorizationKey`, `CollectionThroughput` and `DocumentTemplate` (optional) in `App.config`.

NOTE

Before provisioning collections with high throughput, please refer to the [Pricing Page](#) to estimate the costs per collection. DocumentDB bills storage and throughput independently on an hourly basis, so you can save costs by deleting or lowering the throughput of your DocumentDB collections after testing.

Step 3: Compile and run the console app from the command line. You should see output like the following:

Summary:

Endpoint: https://docdb-scale-demo.documents.azure.com:443/
Collection : db.testdata at 50000 request units per second
Document Template*: Player.json
Degree of parallelism*: 500

DocumentDBBenchmark starting...
Creating database db
Creating collection testdata
Creating metric collection metrics
Retrying after sleeping for 00:03:34.172000
Starting Inserts with 500 tasks
Inserted 661 docs @ 656 writes/s, 6860 RU/s (18B max monthly 1KB reads)
Inserted 6505 docs @ 2668 writes/s, 27962 RU/s (72B max monthly 1KB reads)
Inserted 11756 docs @ 3240 writes/s, 33957 RU/s (88B max monthly 1KB reads)
Inserted 17076 docs @ 3590 writes/s, 37627 RU/s (98B max monthly 1KB reads)
Inserted 22106 docs @ 3748 writes/s, 39281 RU/s (102B max monthly 1KB reads)
Inserted 28430 docs @ 3902 writes/s, 40897 RU/s (106B max monthly 1KB reads)
Inserted 33492 docs @ 3928 writes/s, 41168 RU/s (107B max monthly 1KB reads)
Inserted 38392 docs @ 3963 writes/s, 41528 RU/s (108B max monthly 1KB reads)
Inserted 43371 docs @ 4012 writes/s, 42051 RU/s (109B max monthly 1KB reads)
Inserted 48477 docs @ 4035 writes/s, 42282 RU/s (110B max monthly 1KB reads)
Inserted 53845 docs @ 4088 writes/s, 42845 RU/s (111B max monthly 1KB reads)
Inserted 59267 docs @ 4138 writes/s, 43364 RU/s (112B max monthly 1KB reads)
Inserted 64703 docs @ 4197 writes/s, 43981 RU/s (114B max monthly 1KB reads)
Inserted 70428 docs @ 4216 writes/s, 44181 RU/s (115B max monthly 1KB reads)
Inserted 75868 docs @ 4247 writes/s, 44505 RU/s (115B max monthly 1KB reads)
Inserted 81571 docs @ 4280 writes/s, 44852 RU/s (116B max monthly 1KB reads)
Inserted 86271 docs @ 4273 writes/s, 44783 RU/s (116B max monthly 1KB reads)
Inserted 91993 docs @ 4299 writes/s, 45056 RU/s (117B max monthly 1KB reads)
Inserted 97469 docs @ 4292 writes/s, 44984 RU/s (117B max monthly 1KB reads)
Inserted 99736 docs @ 4192 writes/s, 43930 RU/s (114B max monthly 1KB reads)
Inserted 99997 docs @ 4013 writes/s, 42051 RU/s (109B max monthly 1KB reads)
Inserted 100000 docs @ 3846 writes/s, 40304 RU/s (104B max monthly 1KB reads)

Summary:

Inserted 100000 docs @ 3834 writes/s, 40180 RU/s (104B max monthly 1KB reads)

DocumentDBBenchmark completed successfully.

Step 4 (if necessary): The throughput reported (RU/s) from the tool should be the same or higher than the provisioned throughput of the collection. If not, increasing the DegreeOfParallelism in small increments may help you reach the limit. If the throughput from your client app plateaus, launching multiple instances of the app on the same or different machines will help you reach the provisioned limit across the different instances. If you need help with this step, please, write an email to askdocdb@microsoft.com or fill a support ticket.

Once you have the app running, you can try different [Indexing policies](#) and [Consistency levels](#) to understand their impact on throughput and latency. You can also review the source code and implement similar configurations to your own test suites or production applications.

Next steps

In this article, we looked at how you can perform performance and scale testing with DocumentDB using a .NET console app. Please refer to the links below for additional information on working with DocumentDB.

- [DocumentDB performance testing sample](#)
- [Client configuration options to improve DocumentDB performance](#)
- [Server-side partitioning in DocumentDB](#)
- [DocumentDB collections and performance levels](#)

- [DocumentDB .NET SDK documentation on MSDN](#)
- [DocumentDB .NET samples](#)
- [DocumentDB blog on performance tips](#)

Performance tips for DocumentDB

11/22/2016 • 13 min to read • [Edit on GitHub](#)

Contributors

[mimig](#) • [Theano Petersen](#) • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [jayantacs](#)

Azure DocumentDB is a fast and flexible distributed database that scales seamlessly with guaranteed latency and throughput. You do not have to make major architecture changes or write complex code to scale your database with DocumentDB. Scaling up and down is as easy as making a single API call or [SDK method call](#). However, because DocumentDB is accessed via network calls there are client-side optimizations you can make to achieve peak performance.

So if you're asking "How can I improve my database performance?" consider the following options:

Networking

1. Connection policy: Use direct connection mode

How a client connects to Azure DocumentDB has important implications on performance, especially in terms of observed client-side latency. There are two key configuration settings available for configuring client Connection Policy – the connection *mode* and the [connection protocol](#). The two available modes are:

- a. Gateway Mode (default)
- b. Direct Mode

Gateway Mode is supported on all SDK platforms and is the configured default. If your application runs within a corporate network with strict firewall restrictions, Gateway Mode is the best choice since it uses the standard HTTPS port and a single endpoint. The performance tradeoff, however, is that Gateway Mode involves an additional network hop every time data is read or written to DocumentDB. Because of this, Direct Mode offers better performance due to fewer network hops.

2. Connection policy: Use the TCP protocol

When leveraging Direct Mode, there are two protocol options available:

- TCP
- HTTPS

DocumentDB offers a simple and open RESTful programming model over HTTPS. Additionally, it offers an efficient TCP protocol, which is also RESTful in its communication model and is available through the .NET client SDK. Both Direct TCP and HTTPS use SSL for initial authentication and encrypting traffic. For best performance, use the TCP protocol when possible.

When using TCP in Gateway Mode, TCP Port 443 is the DocumentDB port, and 10250 is the MongoDB API port. When using TCP in Direct Mode, in addition to the Gateway ports, you'll need to ensure the port range between 10000 and 20000 is open because DocumentDB uses dynamic TCP ports. If these ports are not open and you attempt to use TCP, you will receive a 503 Service Unavailable error.

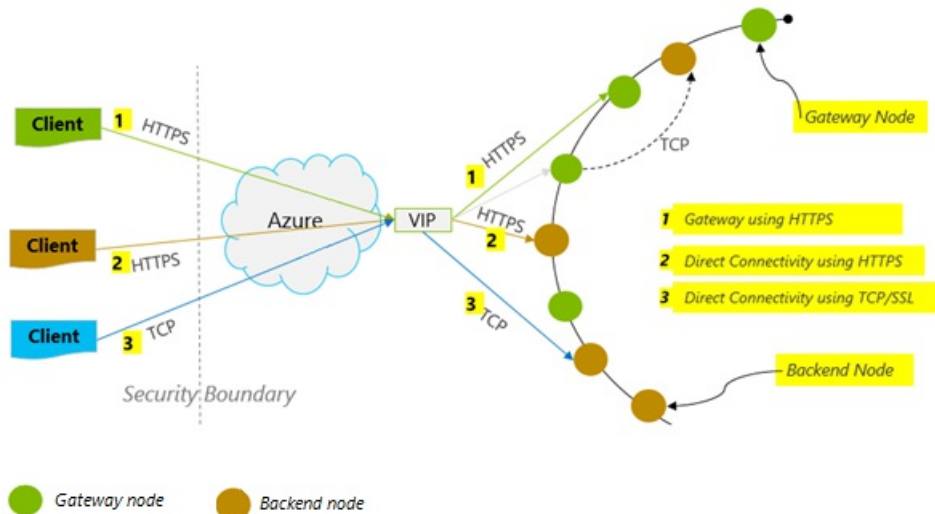
The Connectivity Mode is configured during the construction of the DocumentClient instance with the ConnectionPolicy parameter. If Direct Mode is used, the Protocol can also be set within the ConnectionPolicy parameter.

```

var serviceEndpoint = new Uri("https://contoso.documents.net");
var authKey = new "your authKey from Azure Mngt Portal";
DocumentClient client = new DocumentClient(serviceEndpoint, authKey,
new ConnectionPolicy
{
    ConnectionMode = ConnectionMode.Direct,
    ConnectionProtocol = Protocol.Tcp
});

```

Because TCP is only supported in Direct Mode, if Gateway Mode is used, then the HTTPS protocol is always used to communicate with the Gateway and the Protocol value in the ConnectionPolicy is ignored.



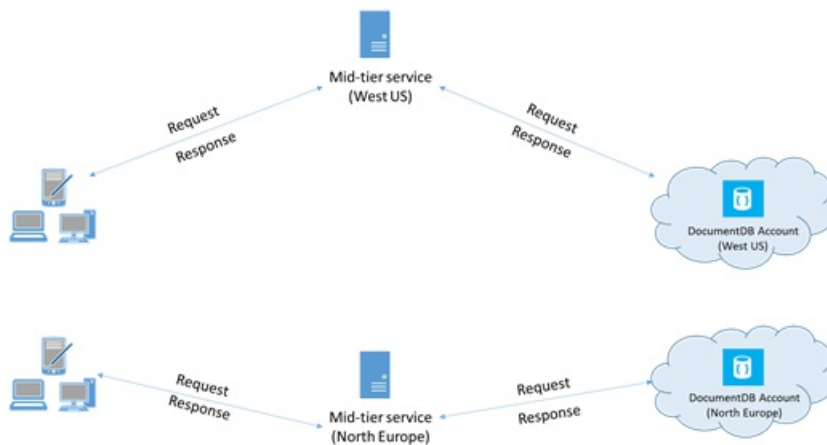
3. Call OpenAsync to avoid startup latency on first request

By default, the first request will have a higher latency because it has to fetch the address routing table. To avoid this startup latency on the first request, you should call `OpenAsync()` once during initialization as follows.

```
await client.OpenAsync();
```

4. Collocate clients in same Azure region for performance

When possible, place any applications calling DocumentDB in the same region as the DocumentDB database. For an approximate comparison, calls to DocumentDB within the same region complete within 1-2 ms, but the latency between the West and East coast of the US is >50 ms. This latency can likely vary from request to request depending on the route taken by the request as it passes from the client to the Azure datacenter boundary. The lowest possible latency is achieved by ensuring the calling application is located within the same Azure region as the provisioned DocumentDB endpoint. For a list of available regions, see [Azure Regions](#).



5. Increase number of threads/tasks

Since calls to DocumentDB are made over the network, you may need to vary the degree of parallelism of your requests so that the client application spends very little time waiting between requests. For example, if you're using .NET's [Task Parallel Library](#), create in the order of 100s of Tasks reading or writing to DocumentDB.

SDK Usage

1. Install the most recent SDK

The DocumentDB SDKs are constantly being improved to provide the best performance. See the [DocumentDB SDK](#) pages to determine the most recent SDK and review improvements.

2. Use a singleton DocumentDB client for the lifetime of your application

Note that each DocumentClient instance is thread-safe and performs efficient connection management and address caching when operating in Direct Mode. To allow efficient connection management and better performance by DocumentClient, it is recommended to use a single instance of DocumentClient per AppDomain for the lifetime of the application.

3. Increase System.Net MaxConnections per host

DocumentDB requests are made over HTTPS/REST by default, and are subjected to the default connection limit per hostname or IP address. You may need to set the MaxConnections to a higher value (100-1000) so that the client library can utilize multiple simultaneous connections to DocumentDB. In the .NET SDK 1.8.0 and above, the default value for [ServicePointManager.DefaultConnectionLimit](#) is 50 and to change the value, you can set the [Documents.Client.ConnectionPolicy.MaxConnectionLimit](#) to a higher value.

4. Tuning parallel queries for partitioned collections

DocumentDB .NET SDK version 1.9.0 and above support parallel queries, which enable you to query a partitioned collection in parallel (see [Working with the SDKs](#) and the related [code samples](#) for more info). Parallel queries are designed to improve query latency and throughput over their serial counterpart. Parallel queries provide two parameters that users can tune to custom-fit their requirements, (a) [MaxDegreeOfParallelism](#): to control the maximum number of partitions than can be queried in parallel, and (b) [MaxBufferedItemCount](#): to control the number of pre-fetched results.

(a) **Tuning [MaxDegreeOfParallelism](#)**: Parallel query works by querying multiple partitions in parallel. However, data from an individual partitioned collect is fetched serially with respect to the query. So, setting the [MaxDegreeOfParallelism](#) to the number of partitions has the maximum chance of achieving the most performant query, provided all other system conditions remain the same. If you don't know the number of partitions, you can set the [MaxDegreeOfParallelism](#) to a high number, and the system will choose the minimum (number of partitions, user provided input) as the [MaxDegreeOfParallelism](#).

It is important to note that parallel queries produce the best benefits if the data is evenly distributed across all partitions with respect to the query. If the partitioned collection is partitioned such a way that all or a majority of the data returned by a query is concentrated in a few partitions (one partition in worst case), then the performance of the query would be bottlenecked by those partitions.

(b) ***Tuning MaxBufferedItemCount***: Parallel query is designed to pre-fetch results while the current batch of results is being processed by the client. The pre-fetching helps in overall latency improvement of a query. MaxBufferedItemCount is the parameter to limit the amount of pre-fetched results. Setting MaxBufferedItemCount to the expected number of results returned (or a higher number) allows the query to receive maximum benefit from pre-fetching.

Note that pre-fetching works the same way irrespective of the MaxDegreeOfParallelism, and there is a single buffer for the data from all partitions.

5. Turn on server-side GC

Reducing the frequency of garbage collection may help in some cases. In .NET, set `gcServer` to true.

6. Implement backoff at RetryAfter intervals

During performance testing, you should increase load until a small rate of requests get throttled. If throttled, the client application should backoff on throttle for the server-specified retry interval. Respecting the backoff ensures that you spend minimal amount of time waiting between retries. Retry policy support is included in Version 1.8.0 and above of the DocumentDB [.NET](#) and [Java](#), version 1.9.0 and above of the [Node.js](#) and [Python](#), and all supported versions of the [.NET Core](#) SDKs. For more information, see [Exceeding reserved throughput limits](#) and [RetryAfter](#).

7. Scale out your client-workload

If you are testing at high throughput levels (>50,000 RU/s), the client application may become the bottleneck due to the machine capping out on CPU or Network utilization. If you reach this point, you can continue to push the DocumentDB account further by scaling out your client applications across multiple servers.

8. Cache document URIs for lower read latency

Cache document URIs whenever possible for the best read performance.

9. Tune the page size for queries/read feeds for better performance

When performing a bulk read of documents using read feed functionality (i.e., ReadDocumentFeedAsync) or when issuing a DocumentDB SQL query, the results are returned in a segmented fashion if the result set is too large. By default, results are returned in chunks of 100 items or 1 MB, whichever limit is hit first.

To reduce the number of network round trips required to retrieve all applicable results, you can increase the page size using x-ms-max-item-count request header to up to 1000. In cases where you need to display only a few results, e.g., if your user interface or application API returns only 10 results a time, you can also decrease the page size to 10 to reduce the throughput consumed for reads and queries.

You may also set the page size using the available DocumentDB SDKs. For example:

```
IQueryable<dynamic> authorResults = client.CreateDocumentQuery(documentCollection.SelfLink, "SELECT p.Author FROM Pages p WHERE p.Title = 'About Seattle'", new FeedOptions { MaxItemCount = 1000 });
```

10. Increase number of threads/tasks

See [Increase number of threads/tasks](#) in the Networking section.

Indexing Policy

1. Use lazy indexing for faster peak time ingestion rates

DocumentDB allows you to specify – at the collection level – an indexing policy, which enables you to choose if you want the documents in a collection to be automatically indexed or not. In addition, you may also choose between synchronous (Consistent) and asynchronous (Lazy) index updates. By default, the index is updated synchronously on each insert, replace, or delete of a document to the collection.

Synchronously mode enables the queries to honor the same [consistency level](#) as that of the document reads without any delay for the index to “catch up”.

Lazy indexing may be considered for scenarios in which data is written in bursts, and you want to amortize the work required to index content over a longer period of time. Lazy indexing also allows you to use your provisioned throughput effectively and serve write requests at peak times with minimal latency. It is important to note, however, that when lazy indexing is enabled, query results will be eventually consistent regardless of the consistency level configured for the DocumentDB account.

Hence, Consistent indexing mode (IndexingPolicy.IndexingMode is set to Consistent) incurs the highest request unit charge per write, while Lazy indexing mode (IndexingPolicy.IndexingMode is set to Lazy) and no indexing (IndexingPolicy.Automatic is set to False) have zero indexing cost at the time of write.

2. Exclude unused paths from indexing for faster writes

DocumentDB’s indexing policy also allows you to specify which document paths to include or exclude from indexing by leveraging Indexing Paths (IndexingPolicy.IncludedPaths and IndexingPolicy.ExcludedPaths).

The use of indexing paths can offer improved write performance and lower index storage for scenarios in which the query patterns are known beforehand, as indexing costs are directly correlated to the number of unique paths indexed. For example, the following code shows how to exclude an entire section of the documents (a.k.a. a subtree) from indexing using the “*” wildcard.

```
var collection = new DocumentCollection { Id = "excludedPathCollection" };
collection.IndexingPolicy.IncludedPaths.Add(new IncludedPath { Path = "/" });
collection.IndexingPolicy.ExcludedPaths.Add(new ExcludedPath { Path = "/nonIndexedContent/*" });
collection = await client.CreateDocumentCollectionAsync(UriFactory.CreateDatabaseUri("db"), excluded);
```

For more information, see [DocumentDB indexing policies](#).

Throughput

1. Measure and tune for lower request units/second usage

DocumentDB offers a rich set of database operations including relational and hierarchical queries with UDFs, stored procedures, and triggers – all operating on the documents within a database collection. The cost associated with each of these operations vary based on the CPU, IO, and memory required to complete the operation. Instead of thinking about and managing hardware resources, you can think of a request unit (RU) as a single measure for the resources required to perform various database operations and service an application request.

[Request units](#) are provisioned for each database account based on the number of capacity units that you purchase. Request unit consumption is evaluated as a rate per second. Applications that exceed the provisioned request unit rate for their account is limited until the rate drops below the reserved level for the account. If your application requires a higher level of throughput, you can purchase additional capacity units.

The complexity of a query impacts how many Request Units are consumed for an operation. The number of predicates, nature of the predicates, number of UDFs, and the size of the source data set all influence the cost of query operations.

To measure the overhead of any operation (create, update, or delete), inspect the `x-ms-request-charge` header (or the equivalent `RequestCharge` property in `ResourceResponse` or `FeedResponse` in the .NET SDK) to measure the number of request units consumed by these operations.

```
// Measure the performance (request units) of writes
ResourceResponse<Document> response = await client.CreateDocumentAsync(collectionSelfLink, myDocument);
Console.WriteLine("Insert of document consumed {0} request units", response.RequestCharge);
// Measure the performance (request units) of queries
IDocumentQuery<dynamic> queryable = client.CreateDocumentQuery(collectionSelfLink,
queryString).AsDocumentQuery();
while (queryable.HasMoreResults)
{
    FeedResponse<dynamic> queryResponse = await queryable.ExecuteNextAsync<dynamic>();
    Console.WriteLine("Query batch consumed {0} request units", queryResponse.RequestCharge);
}
```

The request charge returned in this header is a fraction of your provisioned throughput (i.e., 2000 RUs / second). For example, if the query above returns 1000 1KB documents, the cost of the operation will be 1000. As such, within one second, the server honors only two such requests before throttling subsequent requests. For more information, see [Request units](#) and the [request unit calculator](#).

2. Handle rate limiting/request rate too large

When a client attempts to exceed the reserved throughput for an account, there are no performance degradation at the server and no use of throughput capacity beyond the reserved level. The server will preemptively end the request with `RequestRateTooLarge` (HTTP status code 429) and return the `x-ms-retry-after-ms` header indicating the amount of time, in milliseconds, that the user must wait before reattempting the request.

```
HTTP Status 429,
Status Line: RequestRateTooLarge
x-ms-retry-after-ms :100
```

The SDKs all implicitly catch this response, respect the server-specified `retry-after` header, and retry the request. Unless your account is being accessed concurrently by multiple clients, the next retry will succeed.

If you have more than one client cumulatively operating consistently above the request rate, the default retry count currently set to 9 internally by the client may not suffice; in this case, the client throws a `DocumentClientException` with status code 429 to the application. The default retry count can be changed by setting the `RetryOptions` on the `ConnectionPolicy` instance. By default, the `DocumentClientException` with status code 429 is returned after a cumulative wait time of 30 seconds if the request continues to operate above the request rate. This occurs even when the current retry count is less than the max retry count, be it the default of 9 or a user-defined value.

While the automated retry behavior helps to improve resiliency and usability for the most applications, it might come at odds when doing performance benchmarks, especially when measuring latency. The client-observed latency will spike if the experiment hits the server throttle and causes the client SDK to silently retry. To avoid latency spikes during performance experiments, measure the charge returned by each operation and ensure that requests are operating below the reserved request rate. For more information, see [Request units](#).

3. Design for smaller documents for higher throughput

The request charge (i.e. request processing cost) of a given operation is directly correlated to the size of the document. Operations on large documents cost more than operations for small documents.

Next steps

For a sample application used to evaluate DocumentDB for high-performance scenarios on a few client machines, see [Performance and scale testing with Azure DocumentDB](#).

Also, to learn more about designing your application for scale and high performance, see [Partitioning and scaling in Azure DocumentDB](#).

DocumentDB protocol support for MongoDB

11/15/2016 • 1 min to read • [Edit on GitHub](#)

Contributors

[Andrew Hoh](#) • [Kim Whitlatch](#) (Beyondsoft Corporation) • [Tyson Nevil](#) • [mimig](#) • [Stephen Baron](#)

What is Azure DocumentDB?

Azure DocumentDB is a fully managed NoSQL database service built for fast and predictable performance, high availability, automatic scaling, and ease of development. Its flexible data model, consistent low latencies, and rich query capabilities make it a great fit for web, mobile, gaming, IoT, and many other applications that need seamless scale. Read more in the [DocumentDB introduction](#).

What is DocumentDB protocol support for MongoDB?

DocumentDB databases can now be used as the data store for apps written for MongoDB. Using existing [drivers](#) for MongoDB, applications can easily and transparently communicate with DocumentDB, in many cases by simply changing a connection string. Using this preview functionality, customers can easily build and run applications in the Azure cloud - leveraging DocumentDB's fully managed and scalable NoSQL databases - while continuing to use familiar skills and tools for MongoDB.

DocumentDB protocol support for MongoDB enables the core MongoDB API functions to Create, Read, Update and Delete (CRUD) data as well as query the database. The currently implemented capabilities have been prioritized based on the needs of common platforms, frameworks, tools, and large scale MongoDB customers evaluating Azure for their cloud platform.

Next steps

- Learn how to [create](#) a DocumentDB account with protocol support for MongoDB.
- Learn how to [connect](#) to a DocumentDB account with protocol support for MongoDB.
- Learn how to [use MongoChef](#) with a DocumentDB account with protocol support for MongoDB.
- Explore DocumentDB with protocol support for MongoDB [samples](#).

How to create a DocumentDB account with protocol support for MongoDB using the Azure portal

11/15/2016 • 2 min to read • [Edit on GitHub](#)

Contributors

[Andrew Hoh](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [Kristine Toliver](#) • [mimig](#) • [Stephen Baron](#)

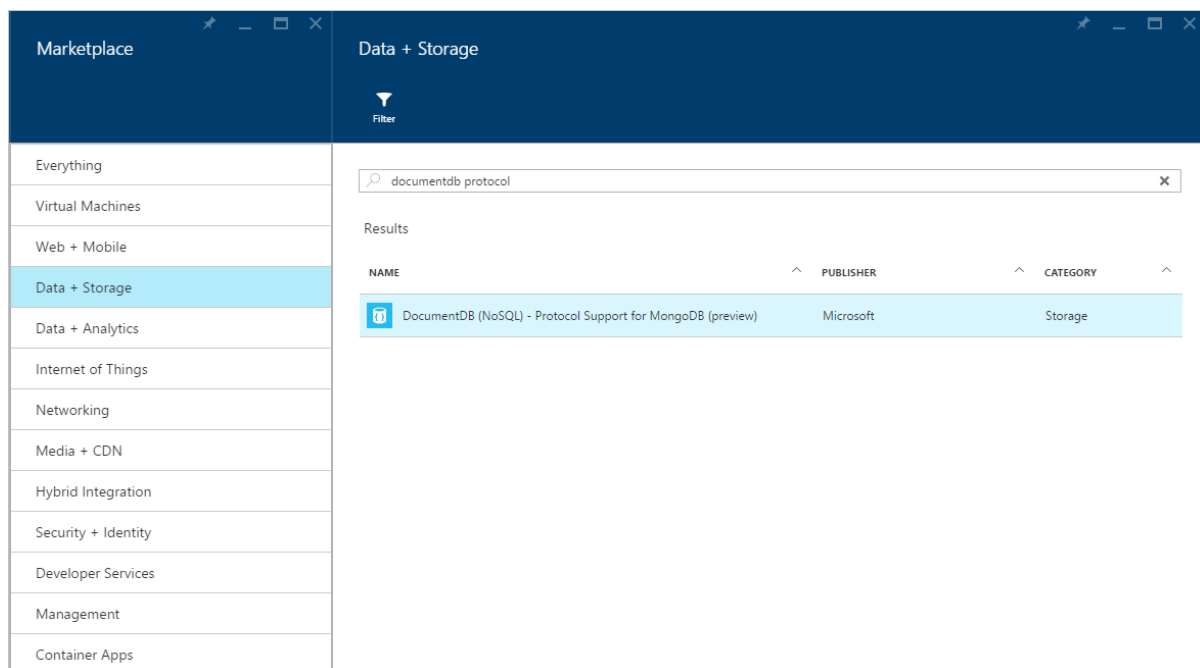
To create an Azure DocumentDB account with protocol support for MongoDB, you must:

- Have an Azure account. You can get a [free Azure account](#) if you don't have one already.

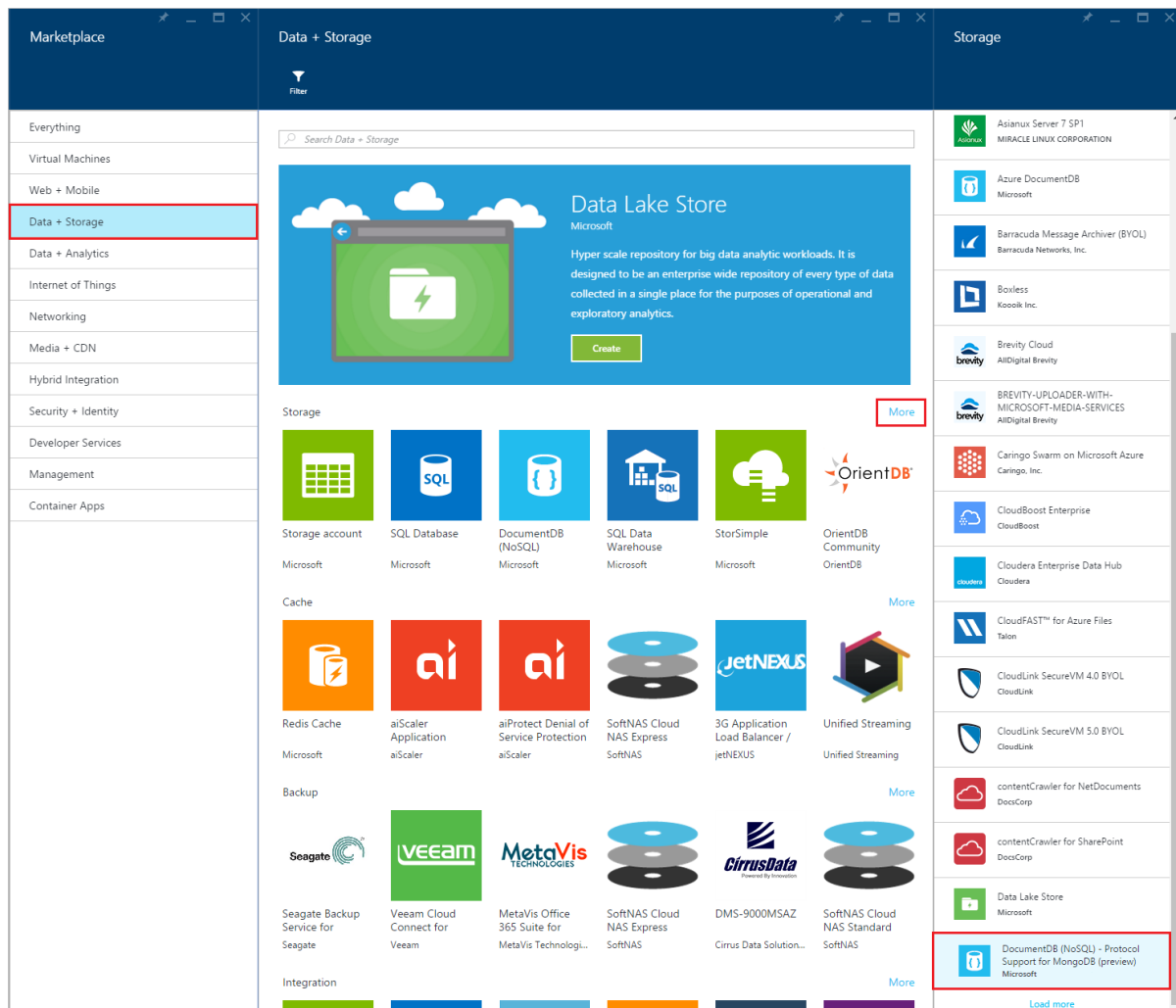
Create the account

To create a DocumentDB account with protocol support for MongoDB, perform the following steps.


1. In a new window, sign in to the [Azure Portal](#).
2. Click **NEW**, click **Data + Storage**, click **See all**, and then search the **Data + Storage** category for "DocumentDB protocol". Click **DocumentDB - Protocol Support for MongoDB**.



3. Alternatively, in the **Data + Storage** category, under **Storage**, click **More**, and then click **Load more** one or more times to display **DocumentDB - Protocol Support for MongoDB**. Click **DocumentDB - Protocol Support for MongoDB**.



4. In the **DocumentDB - Protocol Support for MongoDB (preview)** blade, click **Create** to launch the preview signup process.









DocumentDB - Protocol Support for MongoDB


Microsoft


Microsoft Azure DocumentDB is a fully managed, scalable, queryable NoSQL database service. DocumentDB includes built-in high availability, provides predictable performance, and offers elastic scale-out.


Protocol support (currently in preview) enables applications to seamlessly communicate with DocumentDB using existing drivers for MongoDB.





ONLINE

 sbdocdb001
DOCUMENTDB ACCOUNT

 Add Database

 Document Explorer


 Query Explorer

 Delete

Essentials

Monitoring

Total Requests past week



TOTAL REQUESTS
142

Average Requests per...

No available data.

Operations

All Settings

SBDOCD8001

Search settings

Properties

DocumentDB Quick Start

Keys

Read-Only Keys

Default Consistency

Tags

Users

PUBLISHER

Microsoft

USEFUL LINKS

[Service Overview](#)
[Documentation](#)
[Pricing Details](#)


Create

5. In the DocumentDB account blade, click **Sign up to preview**. Read the information and then click **OK**.

DocumentDB account

with protocol support for MongoDB (PREVIEW)

Sign up to preview today



PUBLIC PREVIEW

DocumentDB with Protocol Support for MongoDB

DocumentDB with Protocol Support for MongoDB. [Learn More](#)

We're currently onboarding a limited set of customers during our Preview.

If you want to be considered for our Preview, please click OK below and we will contact you once you're approved to onboard. Clicking "OK" will put you on the waitlist for DocumentDB with Protocol Support for MongoDB.


The signup will only apply to subscription cabd21e0-70b4-6155-2850-c33ef9163e31

This is a preview feature. By clicking "OK" below, I acknowledge that use of this feature is subject to the preview terms in my license agreement (e.g., the Enterprise Agreement, Microsoft Azure Agreement, or Microsoft Online Subscription Agreement), as well as any applicable Supplemental Terms of Use ([Azure DocumentDB](#)) for Microsoft Azure Previews.

Ok

* Subscription

cabd21e0-70b4-6155-2850-c33ef9163e31



Please accept the below preview terms to request access to the DocumentDB with Protocol Support for MongoDB Preview.

Sign up to preview

Not signed up

- After accepting the preview terms, you will be returned to the create blade. In the **DocumentDB account** blade, specify the desired configuration for the account.


DocumentDB account
with protocol support for MongoDB (PREVIEW)

* ID
contoso123 ✓ documents.azure.com

* Subscription
cabd21e0-70b4-6155-2850-c33ef9163e31 ▼

* Resource Group ⓘ
☒ Create new ☐ Use existing
 New_Resource_Group ✓

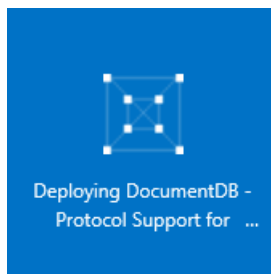
* Location
South Central US ▼

 Provisioning a DocumentDB account will take less than 5 minutes.

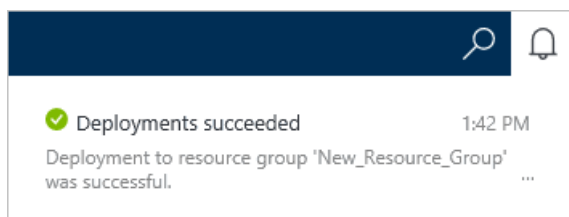
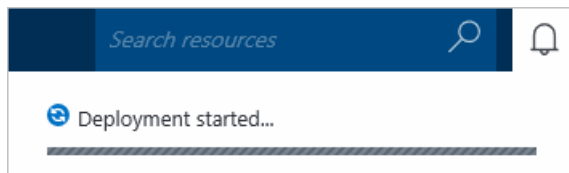
☒ Pin to dashboard

Create


- In the **ID** box, enter a name to identify the account. When the **ID** is validated, a green check mark appears in the **ID** box. The **ID** value becomes the host name within the URI. The **ID** may contain only lowercase letters, numbers, and the '-' character, and must be between 3 and 50 characters. Note that *documents.azure.com* is appended to the endpoint name you choose, the result of which will become your account endpoint.
 - For **Subscription**, select the Azure subscription that you want to use for the account. If your account has only one subscription, that account is selected by default.
 - In **Resource Group**, select or create a resource group for the account. By default, an existing Resource group under the Azure subscription will be chosen. You may, however, choose to select to create a new resource group to which you would like to add the account. For more information, see [Using the Azure portal to manage your Azure resources](#).
 - Use **Location** to specify the geographic location in which to host the account.
 - Optional: Check **Pin to dashboard**. If pinned to dashboard, follow **Step 8** below to view your new account's left-hand navigation.
7. Once the new account options are configured, click **Create**. It can take a few minutes to create the account. If pinned to the dashboard, you can monitor the provisioning progress on the Startboard.



If not pinned to the dashboard, you can monitor your progress from the Notifications hub.



8. To access your new account, click **DocumentDB (NoSQL)** on the left-hand menu. In your list of regular DocumentDB and DocumentDB with Mongo protocol support accounts, click on your new account's name.
9. It is now ready for use with the default settings.



contoso123

DocumentDB account with protocol support for MongoDB

+

 Add Database

→

 Move

↺

 Refresh

🗑️

 Delete Account

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Tutorials and resources

SETTINGS

Connection String

Default consistency

Properties

Locks

Automation script

MONITORING

Alert rules

SUPPORT + TROUBLESHOOTING

New support request

Essentials

Resource group

New_Resource_Group

Subscription Name

Azure Subscription

Location

East US 2

Status

Online

Subscription Id

cabd21e0-70b4-6155-2850-c33ef9163e...

URI

https://contoso123.documents.azure.co...

Databases

ID	COLLECTIONS
admin	2
test	1

Monitoring

Requests

Total RUs

Storage

SBWESTMONGO

0%

CURRENT 103 KiB

THRESHOLD 30 GiB

Next steps

- Learn how to [connect](#) to a DocumentDB account with protocol support for MongoDB.

How to connect to a DocumentDB account with protocol support for MongoDB

11/15/2016 • 1 min to read • [Edit on GitHub](#)

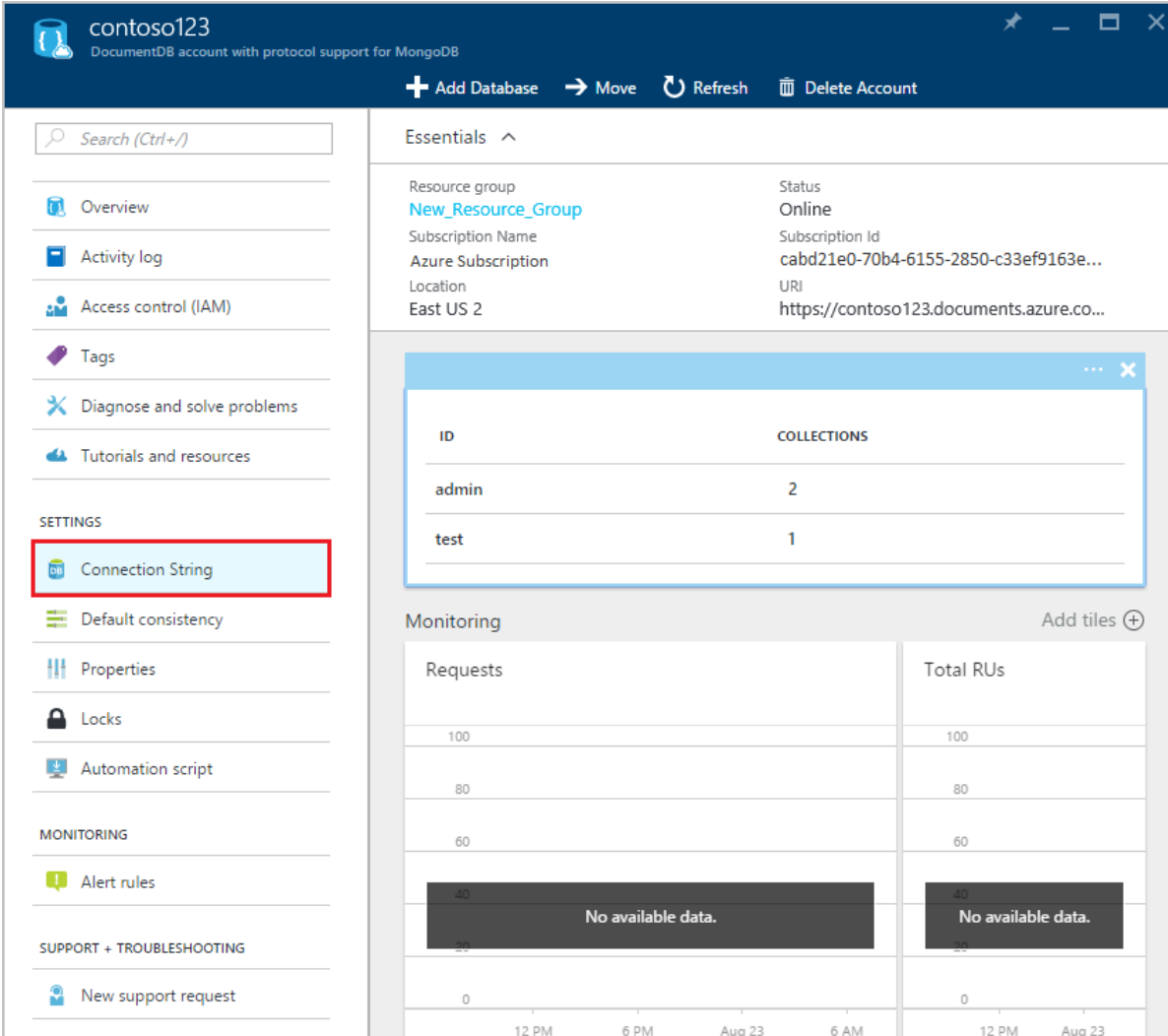
Contributors

[Andrew Hoh](#) • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [mimig](#) • [Stephen Baron](#)

Learn how to connect to an Azure DocumentDB account with protocol support for MongoDB using the standard MongoDB connection string URI format.

Get the account's connection string information

1. In a new window, sign in to the [Azure Portal](#).
2. In the **Left Navigation** bar of the Account Blade, click **Connection String**. To navigate to the **Account Blade**, on the Jumpbar click **More Services**, click **DocumentDB (NoSQL)** , and then select the DocumentDB account with protocol support for MongoDB.

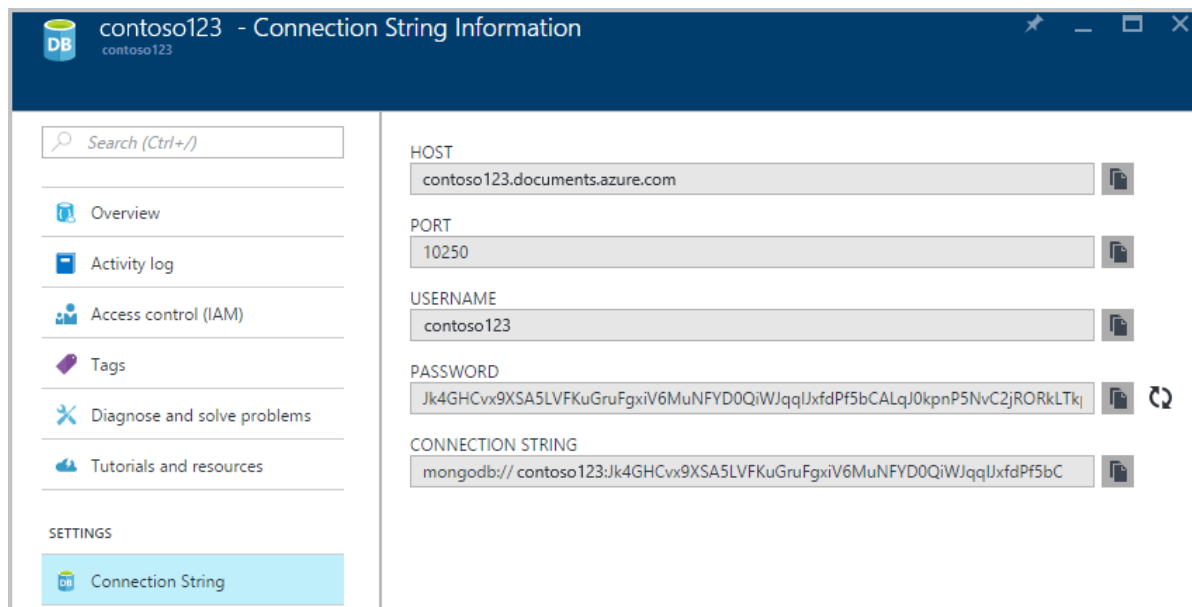


The screenshot shows the Azure Portal interface for a DocumentDB account named 'contoso123'. The left navigation pane is visible, with the 'Connection String' option highlighted in red. The main content area displays the 'Essentials' section, which includes a table of collections and a monitoring section.

ID	COLLECTIONS
admin	2
test	1

The monitoring section shows two charts: 'Requests' and 'Total RUs'. Both charts display a line graph with data points at 12 PM, 6 PM, and 6 AM. The 'Requests' chart shows a peak of 100 requests at 12 PM, while the 'Total RUs' chart shows a peak of 100 RUs at 12 PM. Both charts indicate 'No available data' for the period between 6 PM and 6 AM.

3. The **Connection String Information** blade opens and has all the information necessary to connect to the account using a driver for MongoDB, including a pre-constructed connection string.



Connection string requirements

It is important to note that DocumentDB supports the standard MongoDB connection string URI format, with a couple of specific requirements: DocumentDB accounts require authentication and secure communication via SSL. Thus, the connection string format is:

```
mongodb://username:password@host:port/[database]?ssl=true
```

Where the values of this string are available in the Connection String blade shown above.

- Username (required)
 - DocumentDB account name
- Password (required)
 - DocumentDB account password
- Host (required)
 - FQDN of DocumentDB account
- Port (required)
 - 10250
- Database (optional)
 - The default database used by the connection
- ssl=true (required)

For example, consider the account shown in the Connection String Information above. A valid connection string is:

```
mongodb://contoso123:<password@contoso123.documents.azure.com:10250/mydatabase?ssl=true
```

Connecting with the C# driver for MongoDB

As already mentioned, all DocumentDB accounts require both authentication and secure communication via SSL. While the MongoDB connection string URI format supports an ssl=true query string parameter, working with the MongoDB C# driver requires use of the MongoClientSettings object when creating a MongoClient. Given the account information above, the following code snippet shows how to connect to the account and work with the "Tasks" database.

```
MongoClientSettings settings = new MongoClientSettings();
settings.Server = new MongoServerAddress("contoso123.documents.azure.com", 10250);
settings.UseSsl = true;
settings.SslSettings = new SslSettings();
settings.SslSettings.EnabledSslProtocols = SslProtocols.Tls12;

MongoIdentity identity = new MongoInternalIdentity("Tasks", "contoso123");
MongoIdentityEvidence evidence = new PasswordEvidence("<password>");

settings.Credentials = new List<MongoCredential>()
{
    new MongoCredential("SCRAM-SHA-1", identity, evidence)
};
MongoClient client = new MongoClient(settings);
var database = client.GetDatabase("Tasks",);
```

Next steps

- Learn how to [use MongoChef](#) with a DocumentDB account with protocol support for MongoDB.
- Explore DocumentDB with protocol support for MongoDB [samples](#).

Use MongoChef with a DocumentDB account with protocol support for MongoDB

11/15/2016 • 2 min to read • [Edit on GitHub](#)

Contributors

[Andrew Hoh](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [James Dunn](#) • [mimig](#) • [Stephen Baron](#)

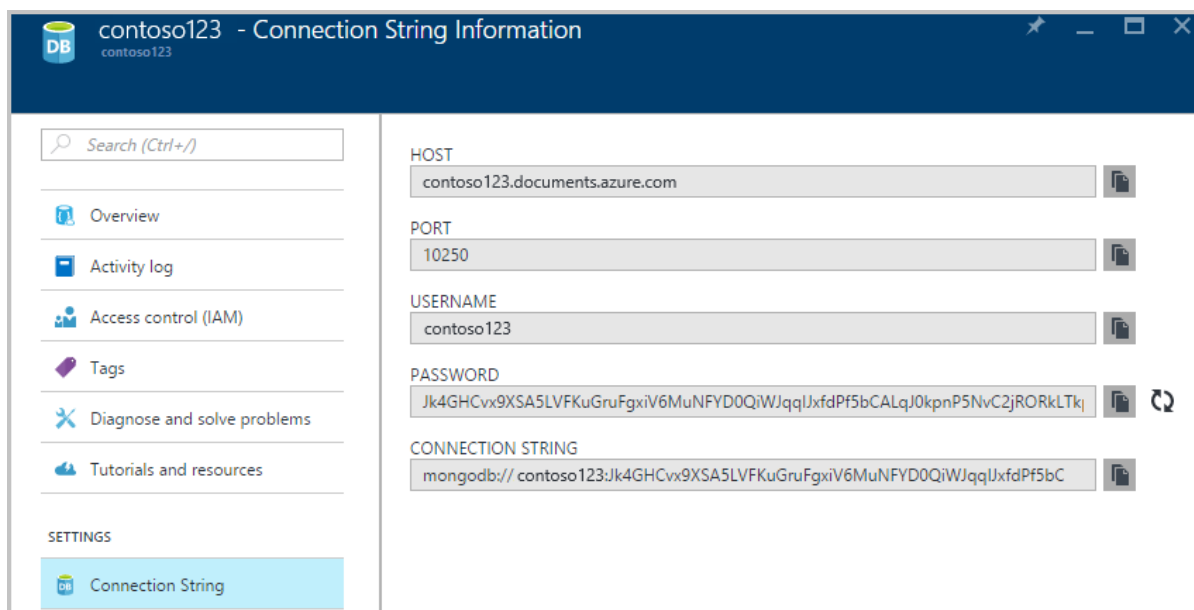
To connect to an Azure DocumentDB account with protocol support for MongoDB using MongoChef, you must:

- Download and install [MongoChef](#)
- Have your DocumentDB account with protocol support for MongoDB [connection string](#) information

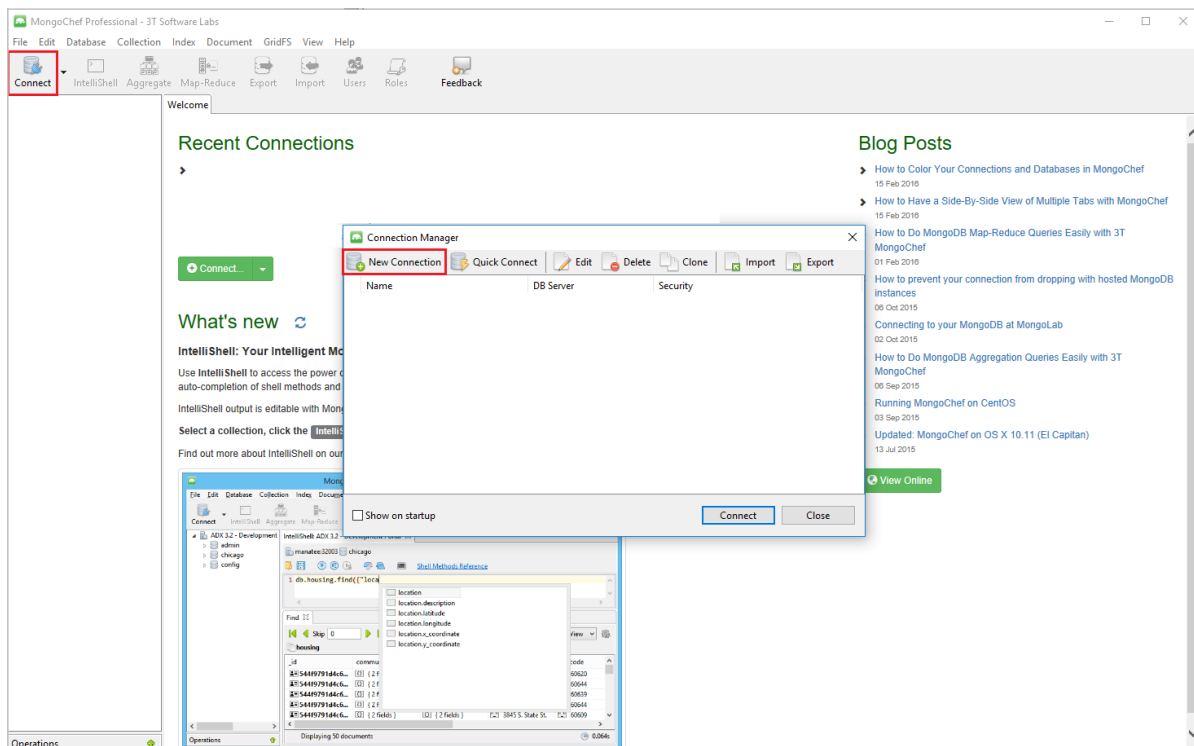
Create the connection in MongoChef

To add your DocumentDB account with protocol support for MongoDB to the MongoChef connection manager, perform the following steps.

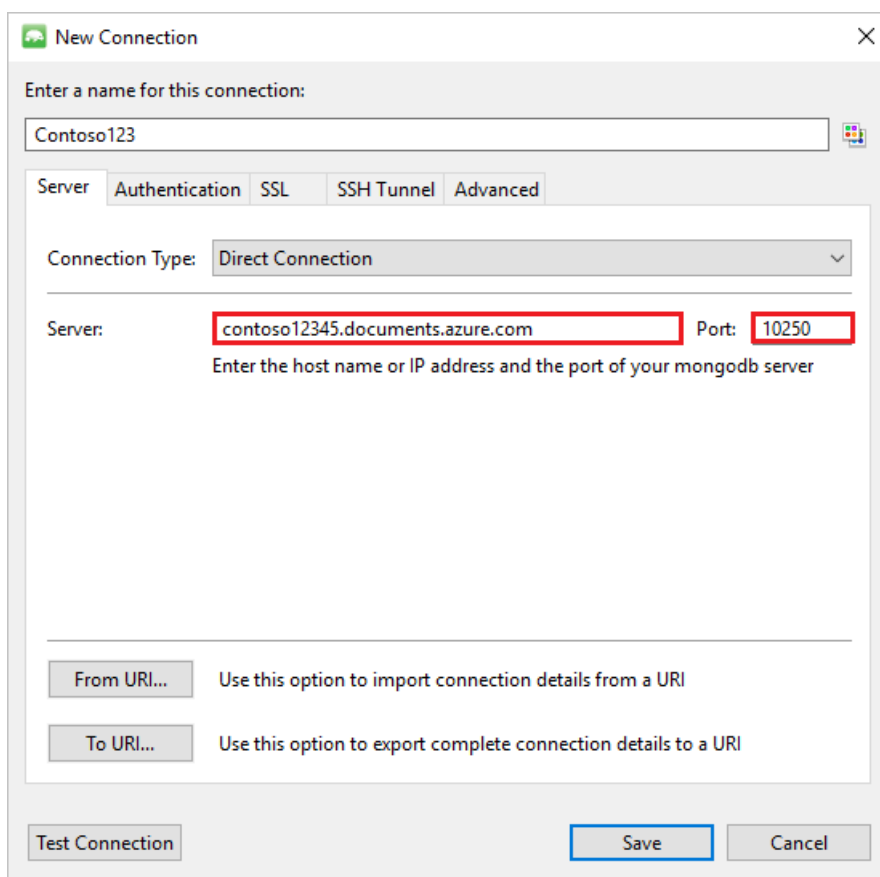
1. Retrieve your DocumentDB with protocol support for MongoDB connection information using the instructions [here](#).



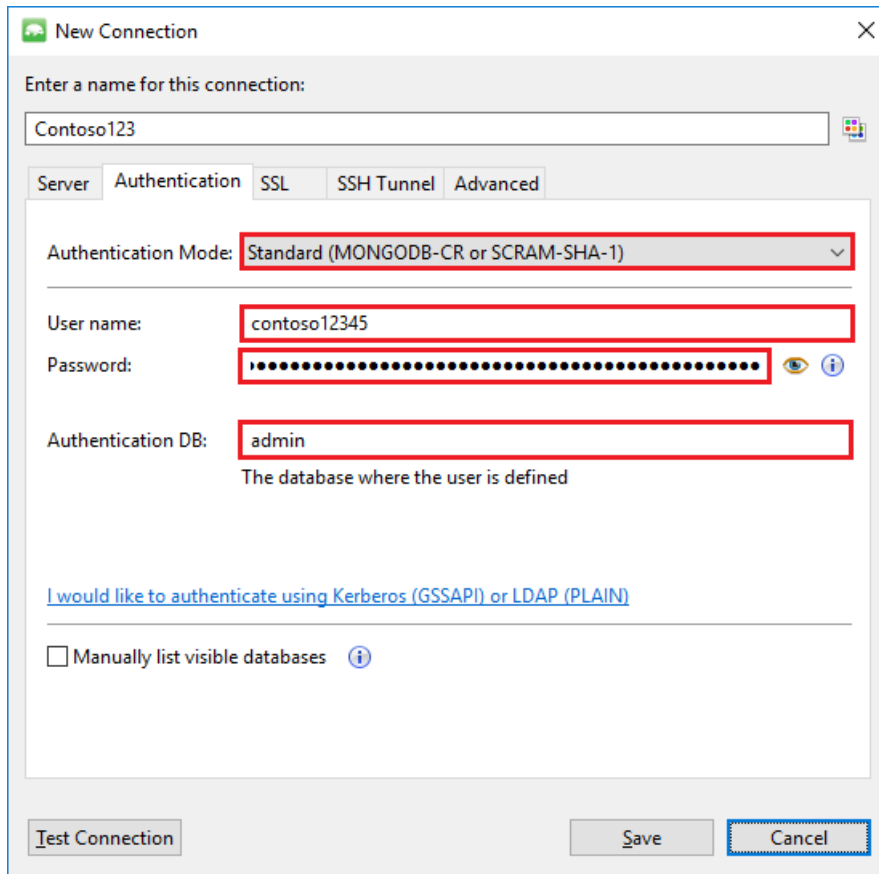
2. Click **Connect** to open the Connection Manager, then click **New Connection**



3. In the **New Connection** window, on the **Server** tab, enter the HOST (FQDN) of the DocumentDB account with protocol support for MongoDB and the PORT.



4. In the **New Connection** window, on the **Authentication** tab, choose Authentication Mode **Standard (MONGODB-CR or SCARM-SHA-1)** and enter the USERNAME and PASSWORD. Accept the default authentication db (admin) or provide your own value.



New Connection

Enter a name for this connection:

Contoso123

Server Authentication SSL SSH Tunnel Advanced

Authentication Mode: Standard (MONGODB-CR or SCRAM-SHA-1)

User name: contoso12345

Password: [Redacted]

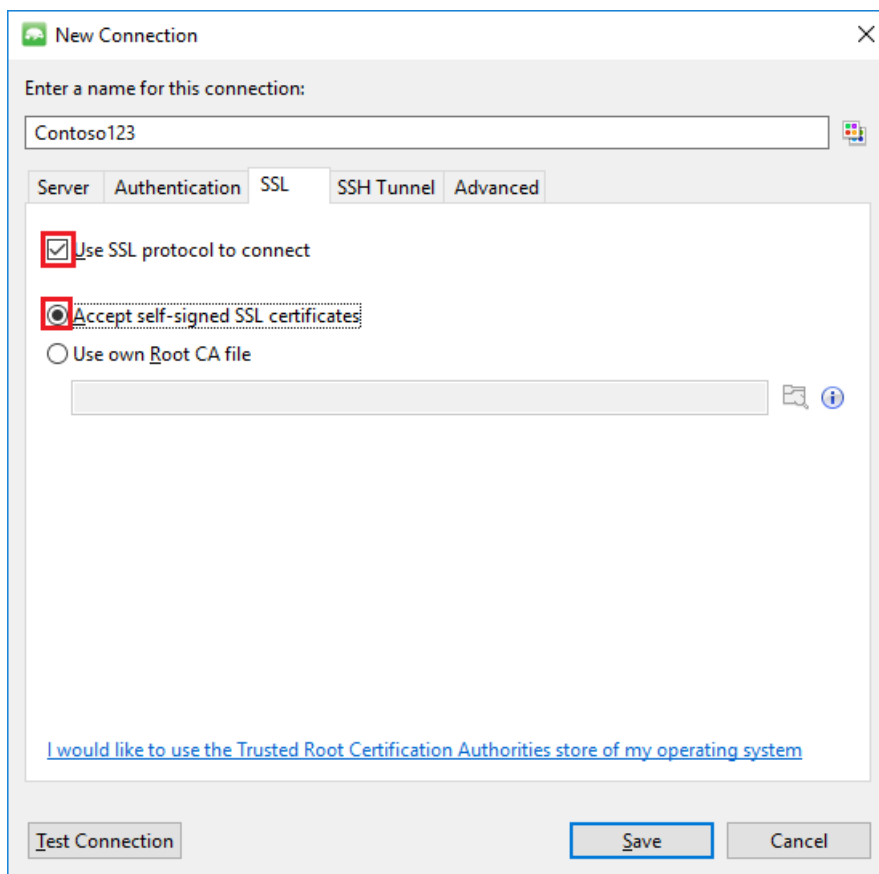
Authentication DB: admin
The database where the user is defined

[I would like to authenticate using Kerberos \(GSSAPI\) or LDAP \(PLAIN\)](#)

☐ Manually list visible databases

Test Connection Save Cancel

- In the **New Connection** window, on the **SSL** tab, check the **Use SSL protocol to connect** check box and the **Accept self-signed SSL certificates** radio button.



New Connection

Enter a name for this connection:

Contoso123

Server Authentication SSL SSH Tunnel Advanced

☒ Use SSL protocol to connect

☒ Accept self-signed SSL certificates

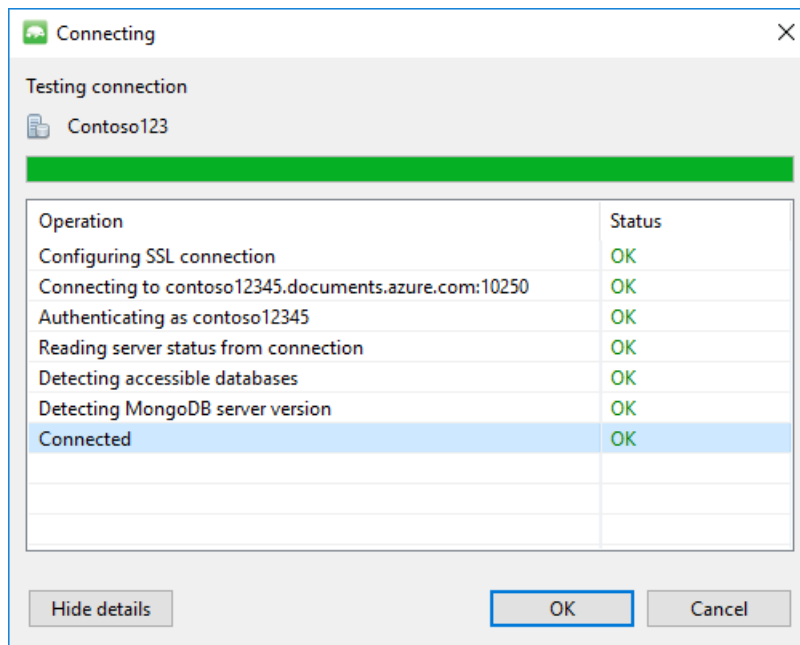
☐ Use own Root CA file

[Redacted]

[I would like to use the Trusted Root Certification Authorities store of my operating system](#)

Test Connection Save Cancel

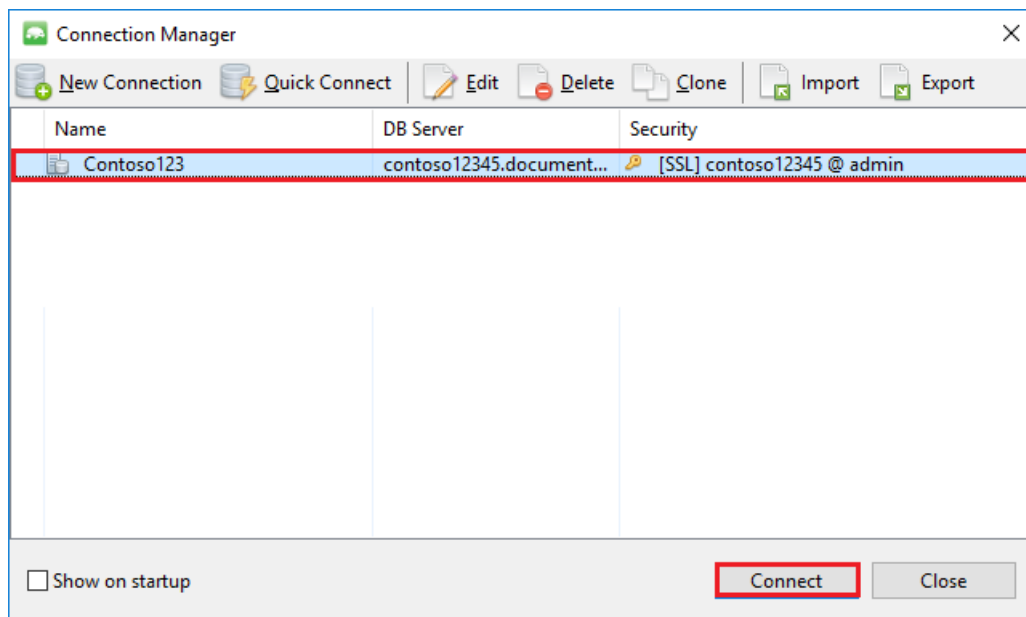
- Click the **Test Connection** button to validate the connection information, click **OK** to return to the New Connection window, and then click **Save**.



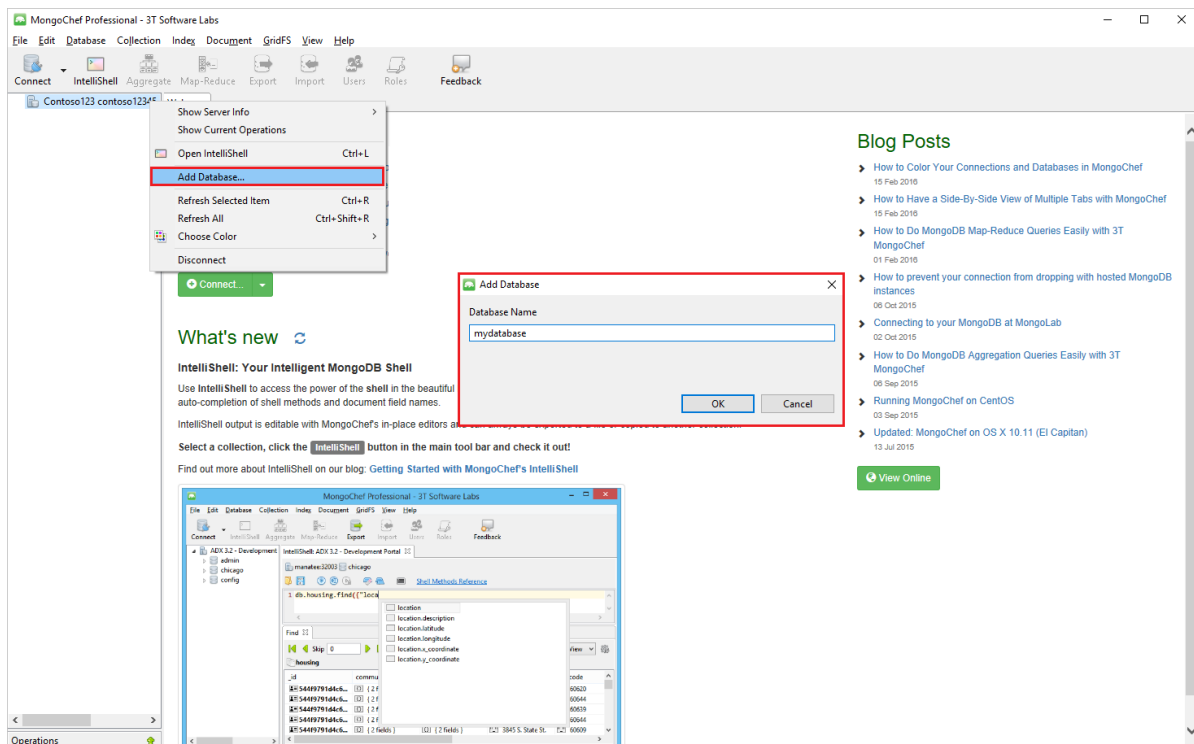
Use MongoChef to create a database, collection, and documents

To create a database, collection, and documents using MongoChef, perform the following steps.

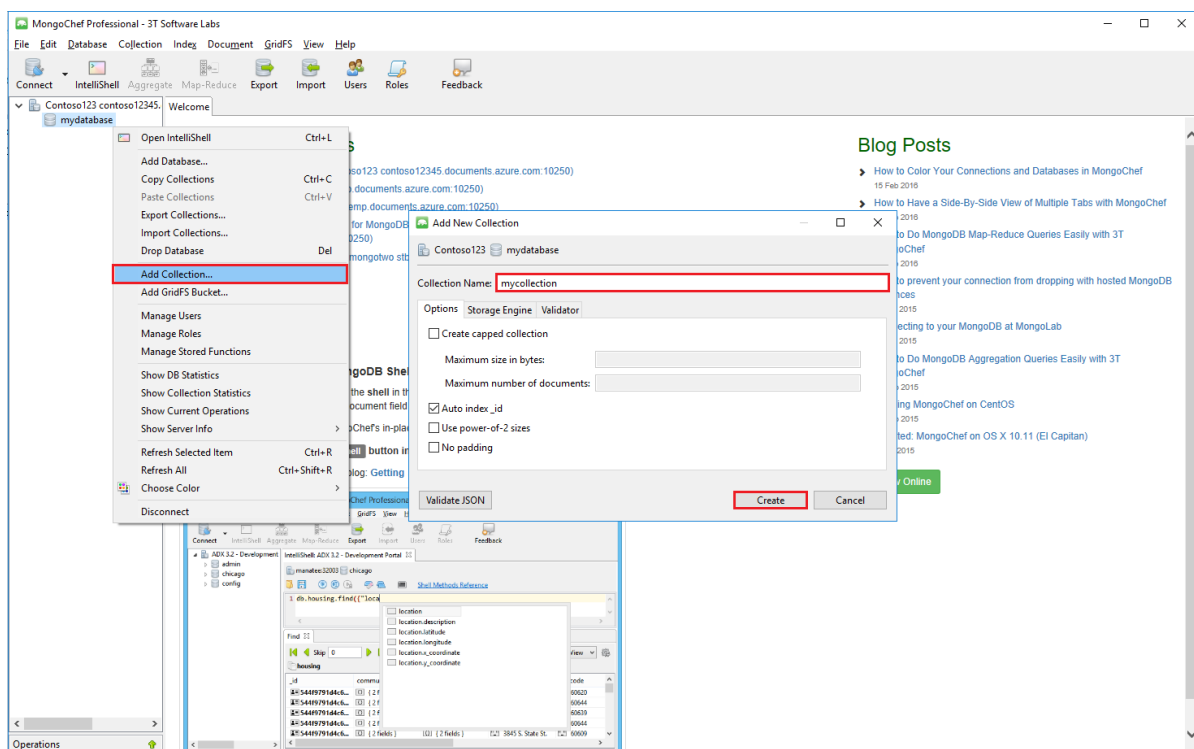
1. In **Connection Manager**, highlight the connection and click **Connect**.



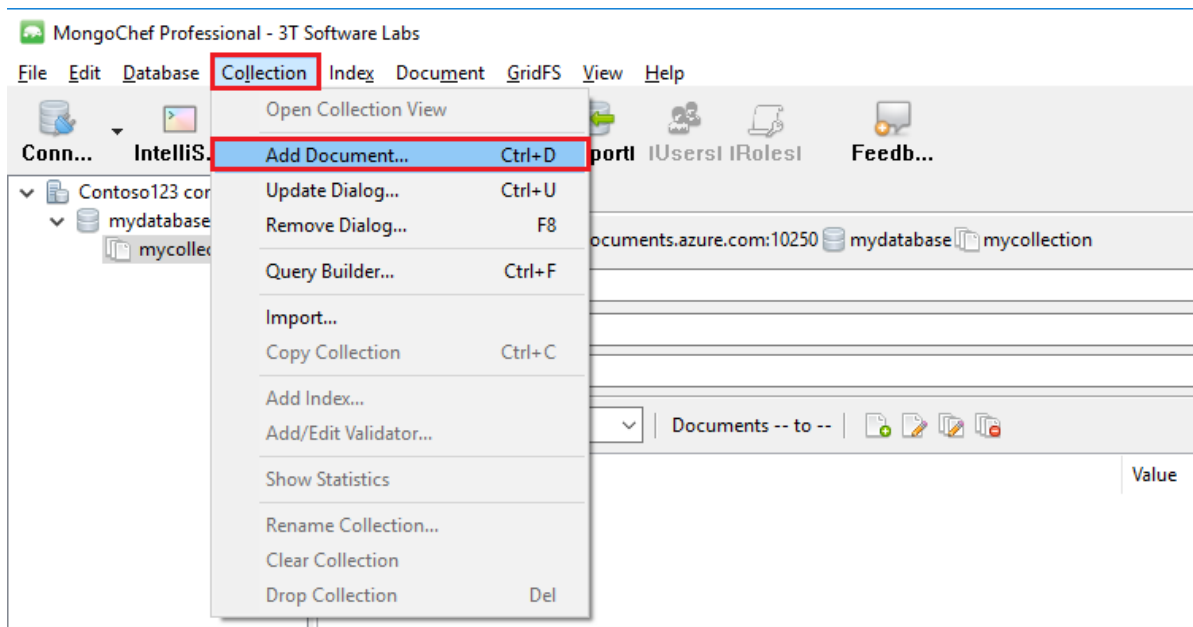
2. Right click the host and choose **Add Database**. Provide a database name and click **OK**.



3. Right click the database and choose **Add Collection**. Provide a collection name and click **Create**.



4. Click the **Collection** menu item, then click **Add Document**.



5. In the Add Document dialog, paste the following and then click **Add Document**.

```
{
  "_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" },
  "isRegistered": true
}
```

6. Add another document, this time with the following content.

```

{
  "_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" },
  "isRegistered": false
}

```

- Execute a sample query. For example, search for families with the last name 'Andersen' and return the parents and state fields.

The screenshot shows the MongoDB Compass interface. The Query tab is active, displaying the query `{lastName: 'Andersen'}`. The Projection tab shows `{parents:1, 'address.state':1}`. The Results tab shows a single document with the following structure:

Key	Value	Type
<code>(1) (1) _id: AndersenFamily</code>	<code>{ 3 fields }</code>	Document
<code>_id</code>	<code>AndersenFamily</code>	String
<code>parents</code>	<code>[2 elements]</code>	Array
<code>0</code>	<code>{ 1 fields }</code>	Object
<code>firstName</code>	<code>Thomas</code>	String
<code>1</code>	<code>{ 1 fields }</code>	Object
<code>firstName</code>	<code>Mary Kay</code>	String
<code>state</code>	<code>WA</code>	String

Next steps

- Explore DocumentDB with protocol support for MongoDB [samples](#).

DocumentDB protocol support for MongoDB examples

11/15/2016 • 2 min to read • [Edit on GitHub](#)

Contributors

[Andrew Hoh](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [James Dunn](#) • [mimig](#) • [Stephen Baron](#)

To use these examples, you must:

- [Create](#) an Azure DocumentDB account with protocol support for MongoDB.
- Retrieve your DocumentDB account with protocol support for MongoDB [connection string](#) information.

Get started with a sample ASP.NET MVC task list application

You can use the [Create a web app in Azure that connects to MongoDB running on a virtual machine](#) tutorial, with minimal modification, to quickly setup a MongoDB application (either locally or published to an Azure web app) that connects to a DocumentDB account with protocol support for MongoDB.

1. Follow the tutorial, with one modification. Replace the Dal.cs code with this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using MyTaskListApp.Models;
using MongoDB.Driver;
using MongoDB.Bson;
using System.Configuration;
using System.Security.Authentication;

namespace MyTaskListApp
{
    public class Dal : IDisposable
    {
        //private MongoServer mongoServer = null;
        private bool disposed = false;

        // To do: update the connection string with the DNS name
        // or IP address of your server.
        //For example, "mongodb://testlinux.cloudapp.net
        private string connectionString = "mongodb://localhost:27017";
        private string userName = "<your user name>";
        private string host = "<your host>";
        private string password = "<your password>";

        // This sample uses a database named "Tasks" and a
        //collection named "TasksList". The database and collection
        //will be automatically created if they don't already exist.
        private string dbName = "Tasks";
        private string collectionName = "TasksList";

        // Default constructor.
        public Dal()
        {
        }

        // Gets all Task items from the MongoDB server.
```

```

public List<MyTask> GetAllTasks()
{
    try
    {
        var collection = GetTasksCollection();
        return collection.Find(new BsonDocument()).ToList();
    }
    catch (MongoConnectionException)
    {
        return new List<MyTask>();
    }
}

// Creates a Task and inserts it into the collection in MongoDB.
public void CreateTask(MyTask task)
{
    var collection = GetTasksCollectionForEdit();
    try
    {
        collection.InsertOne(task);
    }
    catch (MongoCommandException ex)
    {
        string msg = ex.Message;
    }
}

private IMongoCollection<MyTask> GetTasksCollection()
{
    MongoClientSettings settings = new MongoClientSettings();
    settings.Server = new MongoServerAddress(host, 10250);
    settings.UseSsl = true;
    settings.SslSettings = new SslSettings();
    settings.SslSettings.EnabledSslProtocols = SslProtocols.Tls12;

    MongoIdentity identity = new MongoInternalIdentity(dbName, userName);
    MongoIdentityEvidence evidence = new PasswordEvidence(password);

    settings.Credentials = new List<MongoCredential>()
    {
        new MongoCredential("SCRAM-SHA-1", identity, evidence)
    };

    MongoClient client = new MongoClient(settings);
    var database = client.GetDatabase(dbName);
    var todoTaskCollection = database.GetCollection<MyTask>(collectionName);
    return todoTaskCollection;
}

private IMongoCollection<MyTask> GetTasksCollectionForEdit()
{
    MongoClientSettings settings = new MongoClientSettings();
    settings.Server = new MongoServerAddress(host, 10250);
    settings.UseSsl = true;
    settings.SslSettings = new SslSettings();
    settings.SslSettings.EnabledSslProtocols = SslProtocols.Tls12;

    MongoIdentity identity = new MongoInternalIdentity(dbName, userName);
    MongoIdentityEvidence evidence = new PasswordEvidence(password);

    settings.Credentials = new List<MongoCredential>()
    {
        new MongoCredential("SCRAM-SHA-1", identity, evidence)
    };

    MongoClient client = new MongoClient(settings);
    var database = client.GetDatabase(dbName);
    var todoTaskCollection = database.GetCollection<MyTask>(collectionName);
    return todoTaskCollection;
}

```

```

        # region IDisposable

        public void Dispose()
        {
            this.Dispose(true);
            GC.SuppressFinalize(this);
        }

        protected virtual void Dispose(bool disposing)
        {
            if (!this.disposed)
            {
                if (disposing)
                {
                }

                this.disposed = true;
            }

            # endregion
        }
    }
}

```

2. Modify the following variables in the Dal.cs file per your account settings:

```
private string userName = ""; private string host = ""; private string password = "";
```

3. Use the app!

Next steps

- Learn how to [use MongoChef](#) with a DocumentDB account with protocol support for MongoDB.

How to create a DocumentDB NoSQL account using the Azure portal

11/22/2016 • 3 min to read • [Edit on GitHub](#)

Contributors

[mimig](#) • [Theano Petersen](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [v-aljenk](#) • [Jennifer Hubbard](#) • [Ross McAllister](#) • [Dene Hager](#)

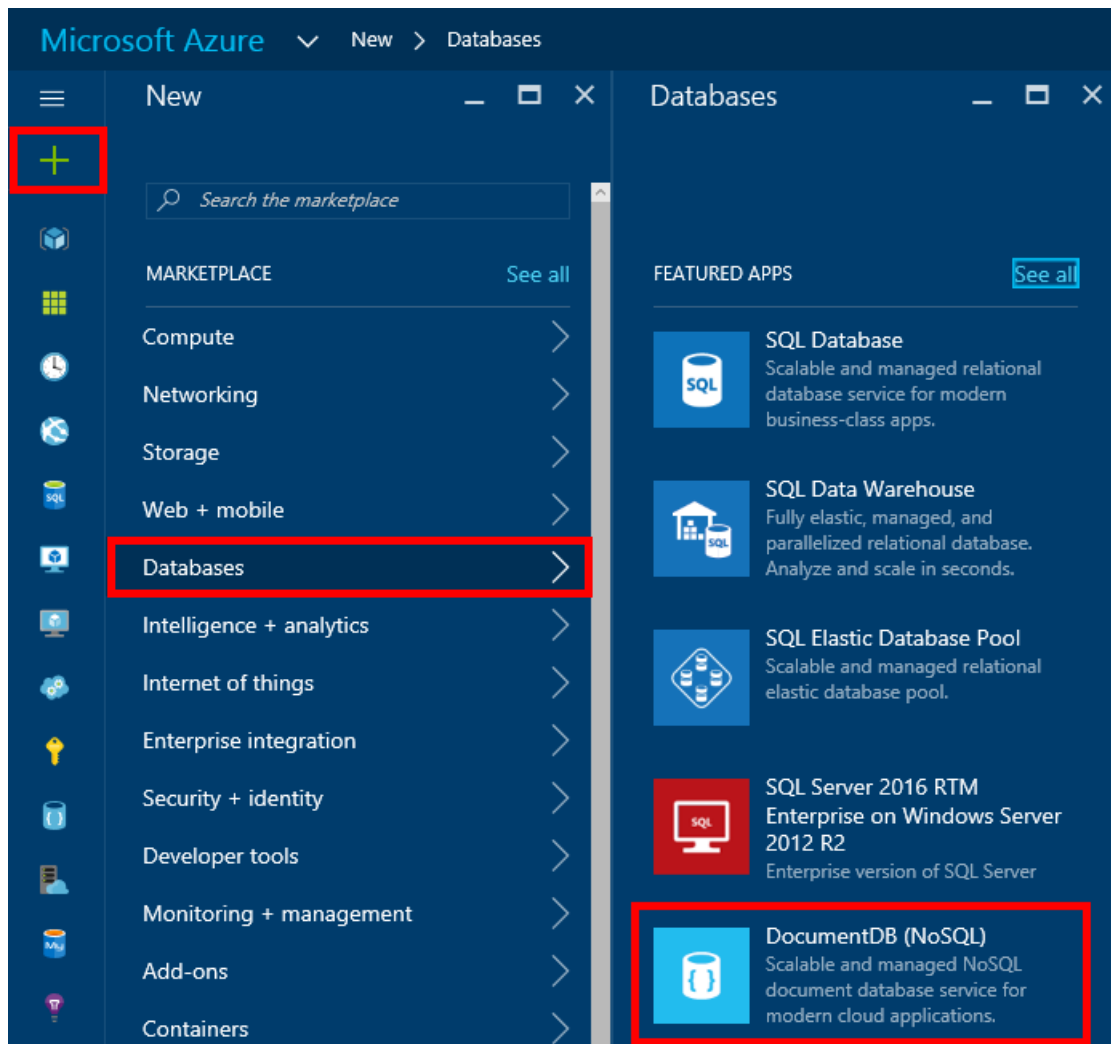
To build a database with Microsoft Azure DocumentDB, you must:

- Have an Azure account. You can get a [free Azure account](#) if you don't have one already.
- Create a DocumentDB account.

You can create a DocumentDB account using either the Azure portal, Azure Resource Manager templates, or Azure command-line interface (CLI). This article shows how to create a DocumentDB account using the Azure portal. To create an account using Azure Resource Manager or Azure CLI, see [Automate DocumentDB database account creation](#).

Are you new to DocumentDB? Watch [this](#) four-minute video by Scott Hanselman to see how to complete the most common tasks in the online portal.

1. Sign in to the [Azure portal](#).
2. In the Jumpbar, click **New**, click **Databases**, and then click **DocumentDB (NoSQL)**.



3. In the **New** account blade, specify the desired configuration for the DocumentDB account.

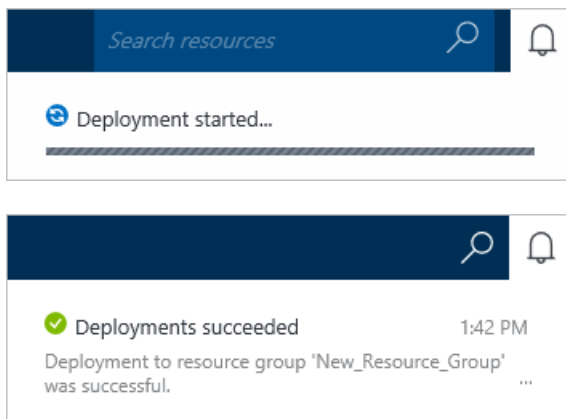
The screenshot shows the 'DocumentDB (NoSQL) New account' blade. The form contains the following fields and values:

- ID:** test-account (with a green checkmark indicating validation). The domain documents.azure.com is shown to the right.
- NoSQL API:** DocumentDB (selected) and MongoDB (available).
- Subscription:** Visual Studio Ultimate with MSDN (selected from a dropdown).
- Resource Group:** Create new (selected) and Use existing (available). The name test-resource-group is entered in the text box.
- Location:** West US (selected from a dropdown).
- Pin to dashboard:** An unchecked checkbox.
- Buttons:** Create and Automation options.

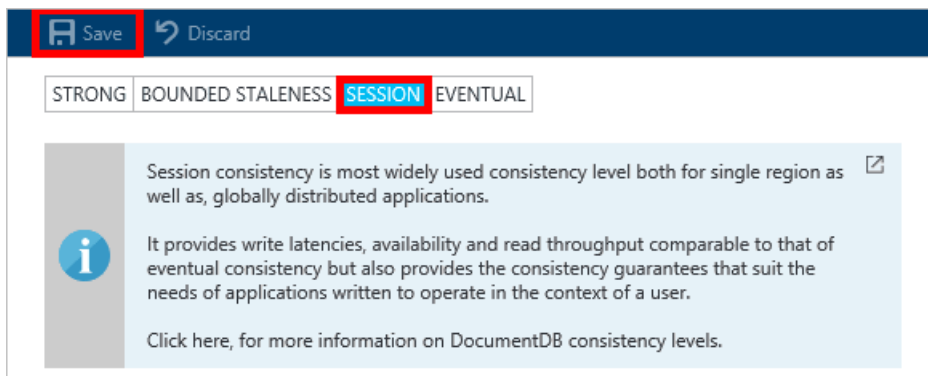
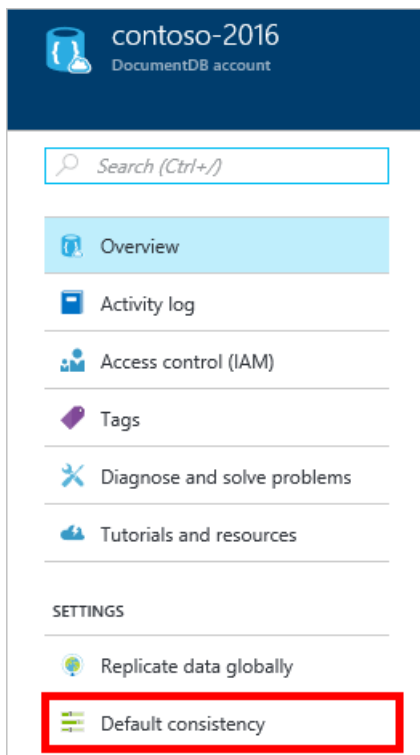
- In the **ID** box, enter a name to identify the DocumentDB account. When the **ID** is validated, a green check mark appears in the **ID** box. The **ID** value becomes the host name within the URI. The **ID** may

contain only lowercase letters, numbers, and the '-' character, and must be between 3 and 50 characters. Note that *documents.azure.com* is appended to the endpoint name you choose, the result of which becomes your DocumentDB account endpoint.

- In the **NoSQL API** box, select the programming model to use:
 - **DocumentDB**: The DocumentDB API is available via .NET, Java, Node.js, Python and JavaScript [SDKs](#), as well as HTTP [REST](#), and offers programmatic access to all the DocumentDB functionality.
 - **MongoDB**: DocumentDB also offers [protocol-level support](#) for **MongoDB** APIs. When you choose the MongoDB API option, you can use existing MongoDB SDKs and [tools](#) to talk to DocumentDB. You can [move](#) your existing MongoDB apps to use DocumentDB, with [no code changes needed](#), and take advantage of a fully managed database as a service, with limitless scale, global replication, and other capabilities.
 - For **Subscription**, select the Azure subscription that you want to use for the DocumentDB account. If your account has only one subscription, that account is selected by default.
 - In **Resource Group**, select or create a resource group for your DocumentDB account. By default, a new resource group is created. For more information, see [Using the Azure portal to manage your Azure resources](#).
 - Use **Location** to specify the geographic location in which to host your DocumentDB account.
4. Once the new DocumentDB account options are configured, click **Create**. To check the status of the deployment, check the Notifications hub.



5. After the DocumentDB account is created, it is ready for use with the default settings. The default consistency of the DocumentDB account is set to **Session**. You can adjust the default consistency by clicking **Default Consistency** in the resource menu. To learn more about the consistency levels offered by DocumentDB, see [Consistency levels in DocumentDB](#).



Next steps

Now that you have a DocumentDB account, the next step is to create a DocumentDB collection and database.

You can create a new collection and database by using one of the following:

- The Azure portal, as described in [Create a DocumentDB collection using the Azure portal](#).
- The all-inclusive tutorials, which include sample data: [.NET](#), [.NET MVC](#), [Java](#), [Node.js](#), or [Python](#).
- The [.NET](#), [Node.js](#), or [Python](#) sample code available in GitHub.
- The [.NET](#), [.NET Core](#), [Node.js](#), [Java](#), [Python](#), and [REST](#) SDKs.

After creating your database and collection, you need to [add documents](#) to the collections.

After you have documents in a collection, you can use [DocumentDB SQL](#) to [execute queries](#) against your documents. You can execute queries by using the [Query Explorer](#) in the portal, the [REST API](#), or one of the [SDKs](#).

Learn more

To learn more about DocumentDB, explore these resources:

- [Learning path for DocumentDB](#)
- [DocumentDB hierarchical resource model and concepts](#)

How to create a DocumentDB collection and database using the Azure portal

11/15/2016 • 5 min to read • [Edit on GitHub](#)

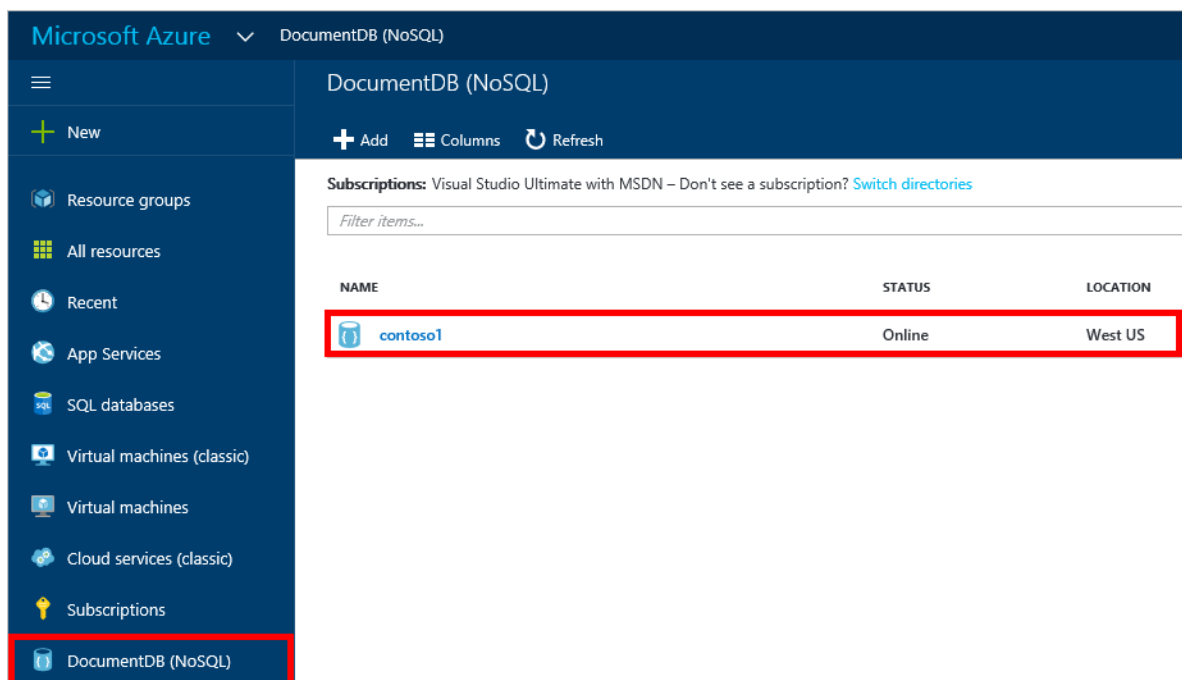
Contributors

mimig • Theano Petersen • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • carolinacmoravia • Carolyn Gronlund • arramac • v-aljenk • Jennifer Hubbard • Dene Hager

To use Microsoft Azure DocumentDB, you must have a [DocumentDB account](#), a database, a collection, and documents. This topic describes how to create a DocumentDB collection in the Azure portal.

Not sure what a collection is? See [What is a DocumentDB collection?](#)

1. In the [Azure portal](#), in the Jumpbar, click **DocumentDB (NoSQL)**, and then in the **DocumentDB (NoSQL)** blade, select the account in which to add a collection. If you don't have any accounts listed, you'll need to [create a DocumentDB account](#).



If **DocumentDB (NoSQL)** is not visible in the Jumpbar, click **More Services** and then click **DocumentDB (NoSQL)**. If you don't have any accounts listed, you'll need to [create a DocumentDB account](#).

2. In the **DocumentDB account** blade for the selected account, click **Add Collection**.



3. In the **Add Collection** blade, in the ID box, enter the ID for your new collection. Collection names must be between 1 and 255 characters, and cannot contain `/ \ # ?` or a trailing space. When the name is validated, a green check mark appears in the ID box.

Add Collection

* Collection Id ⓘ
contoso1 ✓

PRICING TIER ⓘ
Standard >

PARTITIONING MODE ⓘ
Single Partition Partitioned

THROUGHPUT CAPACITY ⓘ
400 - 10000 RU/s *

STORAGE CAPACITY ⓘ
10 GB *

* For more capacity use Partitioned mode.

* DATABASE ⓘ
☒ Create New ☐ Use existing

contoso2016 ✓

OK

4. By default, **Pricing Tier** is set to **Standard** so that you can customize the throughput and storage for your collection. For more information about the pricing tier, see [Performance levels in DocumentDB](#).
5. Select a **Partitioning mode** for the collection, either **Single Partition** or **Partitioned**.

A **single partition** has a reserved storage capacity of 10GB, and can have throughput levels from 400-10,000 request units/second (RU/s). One RU corresponds to the throughput of a read of a 1KB document. For more information about request units, see [Request units](#).

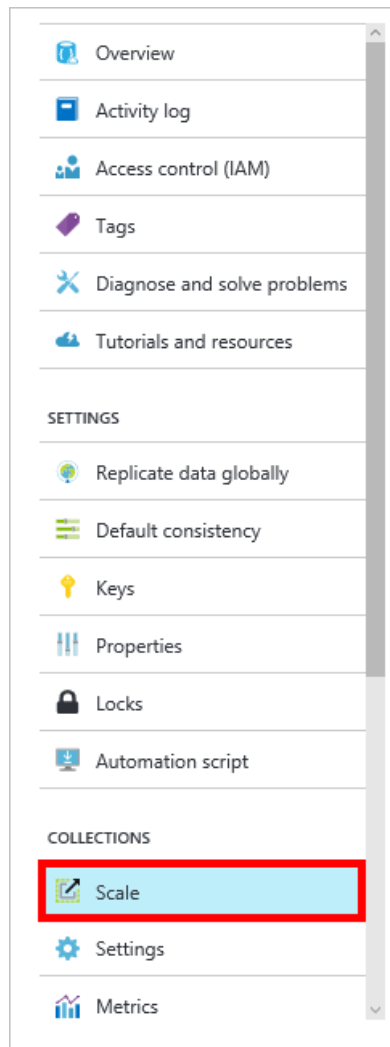
A **partitioned collection** can scale to handle an unlimited amount of storage over multiple partitions, and can have throughput levels starting at 10,100 RU/s. In the portal, the largest storage you can reserve is 250 GB, and the most throughput you can reserve is 250,000 RU/s. To increase either quota, file a request as described in [Request increased DocumentDB account quotas](#). For more information about partitioned collections, see [Single Partition and Partitioned Collections](#).

By default, the throughput for a new single partition collection is set to 1000 RU/s with a storage capacity of 10 GB. For a partitioned collection, the collection throughput is set to 10100 RU/s with a storage capacity of 250 GB. You can change the throughput and storage for the collection after the collection is created.

6. If you are creating a partitioned collection, select the **Partition Key** for the collection. Selecting the correct partition key is important in creating a performant collection. For more information on selecting a partition key, see [Designing for partitioning](#).
7. In the **Database** blade, either create a new database or use an existing one. Database names must be between 1 and 255 characters, and cannot contain `/ \ # ?` or a trailing space. To validate the name, click outside the text box. When the name is validated, a green check mark appears in the box.
8. Click **OK** at the bottom of the screen to create the new collection.
9. The new collection now appears in the **Collections** lens on the **Overview** blade.

ID	DATABASE	THROUGHPUT	PRICING TIER
contoso1	contoso2016	1000	Standard

10. **Optional:** To modify the throughput of collection in the portal, click **Scale** on the Resource menu.



What is a DocumentDB collection?

A collection is a container of JSON documents and the associated JavaScript application logic. A collection is a billable entity, where the [cost](#) is determined by the provisioned throughput of the collection. Collections can span one or more partitions/servers and can scale to handle practically unlimited volumes of storage or throughput.

Collections are automatically partitioned into one or more physical servers by DocumentDB. When you create a collection, you can specify the provisioned throughput in terms of request units per second and a partition key property. The value of this property will be used by DocumentDB to distribute documents among partitions and route requests like queries. The partition key value also acts as the transaction boundary for stored procedures and triggers. Each collection has a reserved amount of throughput specific to that collection, which is not shared with other collections in the same account. Therefore, you can scale out your application both in terms of storage and throughput.

Collections are not the same as tables in relational databases. Collections do not enforce schema, in fact DocumentDB does not enforce any schemas, it's a schema-free database. Therefore you can store different types of documents with diverse schemas in the same collection. You can choose to use collections to store objects of a

single type like you would with tables. The best model depends only on how the data appears together in queries and transactions.

Other ways to create a DocumentDB collection

Collections do not have to be created using the portal, you can also create them using the [DocumentDB SDKs](#) and the REST API.

- For a C# code sample, see the [C# collection samples](#).
- For a Node.js code sample, see the [Node.js collection samples](#).
- For a Python code sample, see [Python collection samples](#).
- For a REST API sample, see [Create a Collection](#).

Troubleshooting

If **Add Collection** is disabled in the Azure portal, that means your account is currently disabled, which normally occurs when all the benefits credits for the month are used.

Next steps

Now that you have a collection, the next step is to add documents or import documents into the collection. When it comes to adding documents to a collection, you have a few choices:

- You can [add documents](#) by using the Document Explorer in the portal.
- You can [import documents and data](#) by using the DocumentDB Data Migration Tool, which enables you to import JSON and CSV files, as well as data from SQL Server, MongoDB, Azure Table storage, and other DocumentDB collections.
- Or you can add documents by using one of the [DocumentDB SDKs](#). DocumentDB has .NET, Java, Python, Node.js, and JavaScript API SDKs. For C# code samples showing how to work with documents by using the DocumentDB .NET SDK, see the [C# document samples](#). For Node.js code samples showing how to work with documents by using the DocumentDB Node.js SDK, see the [Node.js document samples](#).

After you have documents in a collection, you can use [DocumentDB SQL](#) to [execute queries](#) against your documents by using the [Query Explorer](#) in the portal, the [REST API](#), or one of the [SDKs](#).

How to perform DocumentDB global database replication using the Azure portal

11/15/2016 • 2 min to read • [Edit on GitHub](#)

Contributors

[mimig](#) • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#)

Learn how to use the Azure portal to replicate data in multiple regions for global availability of data in Azure DocumentDB.

For information about how global database replication works in DocumentDB, see [Distribute data globally with DocumentDB](#). For information about performing global database replication programmatically, see [Developing with multi-region DocumentDB accounts](#).

NOTE

Global distribution of DocumentDB databases is generally available and automatically enabled for any newly created DocumentDB accounts. We are working to enable global distribution on all existing accounts, but in the interim, if you want global distribution enabled on your account, please [contact support](#) and we'll enable it for you now.

Add global database regions

DocumentDB is available in most [Azure regions](#). After selecting the default consistency level for your database account, you can associate one or more regions (depending on your choice of default consistency level and global distribution needs).

1. In the [Azure portal](#), in the Jumpbar, click **DocumentDB Accounts**.
2. In the **DocumentDB Account** blade, select the database account to modify.
3. In the account blade, click **Replicate data globally** from the menu.
4. In the **Replicate data globally** blade, select the regions to add or remove, and then click **Save**. There is a cost to adding regions, see the [pricing page](#) or the [Distribute data globally with DocumentDB](#) article for more information.

azure-documentdb - Replicate data globally

Search (Ctrl+/)

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Tutorials and resources

SETTINGS

Replicate data globally

Default consistency

Keys

Properties

Locks

Automation script

MONITORING

Alert rules

SUPPORT + TROUBLESHOOTING

New support request

Click on a location to add or remove regions from your DocumentDB account.

* Each region is billable based on the throughput and storage for the account. [Learn more](#)

WRITE REGION

South Central US

READ REGIONS

The account has no read regions.

Selecting global database regions

When configuring two or more regions, it is recommended that regions are selected based on the region pairs described in the [Business continuity and disaster recovery \(BCDR\): Azure Paired Regions](#) article.

Specifically, when configuring to multiple regions, make sure to select the same number of regions (+/-1 for odd/even) from each of the paired region columns. For example, if you want to deploy to four US regions, you select two US regions from the left column and two from the right. So, the following would be an appropriate set: West US, East US, North Central US, and South Central US.

This guidance is important to follow when only two regions are configured for disaster recovery scenarios. For more than two regions, following this guidance is good practice, but not critical as long as some of the selected regions adhere to this pairing.

Next steps

Learn how to manage the consistency of your globally replicated account by reading [Consistency levels in DocumentDB](#).

For information about how global database replication works in DocumentDB, see [Distribute data globally with DocumentDB](#). For information about programmatically replicating data in multiple regions, see [Developing with multi-region DocumentDB accounts](#).

View, edit, create, and upload JSON documents using DocumentDB Document Explorer

11/15/2016 • 5 min to read • [Edit on GitHub](#)

Contributors

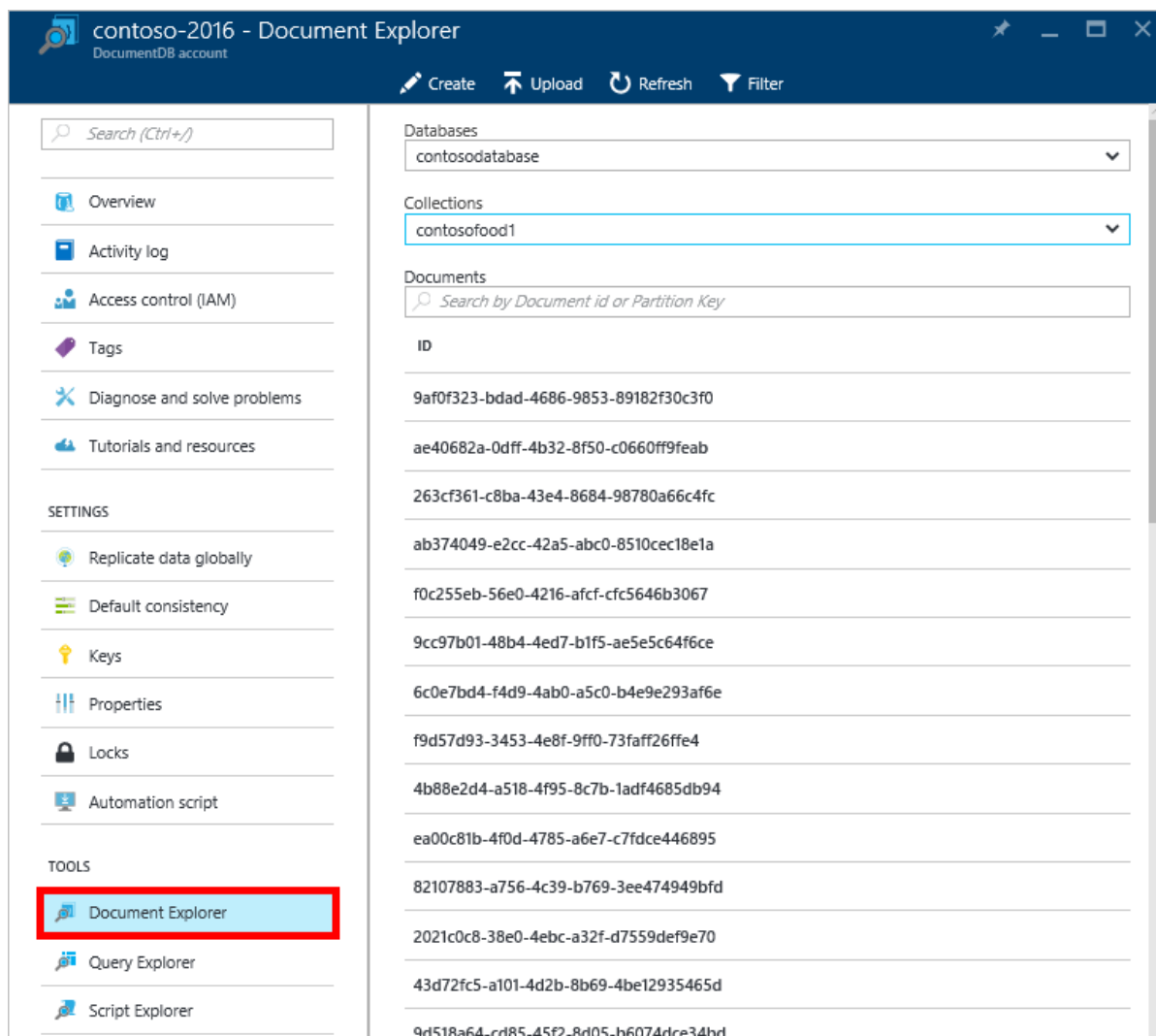
Kirill Gavrylyuk • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • mimig • Andrew Hoh • v-aljenk • Stephen Baron

This article provides an overview of the [Microsoft Azure DocumentDB](#) Document Explorer, an Azure portal tool that enables you to view, edit, create, upload, and filter JSON documents with DocumentDB.

Note that Document Explorer is not enabled on DocumentDB accounts with protocol support for MongoDB. This page will be updated when this feature is enabled.

Launch Document Explorer

1. In the Azure portal, in the Jumpbar, click **DocumentDB (NoSQL)**. If **DocumentDB (NoSQL)** is not visible, click **More Services** and then click **DocumentDB (NoSQL)**.
2. Select the account name.
3. In the resource menu, click **Document Explorer**.



In the **Document Explorer** blade, the **Databases** and **Collections** drop-down lists are pre-populated depending on the context in which you launched Document Explorer.

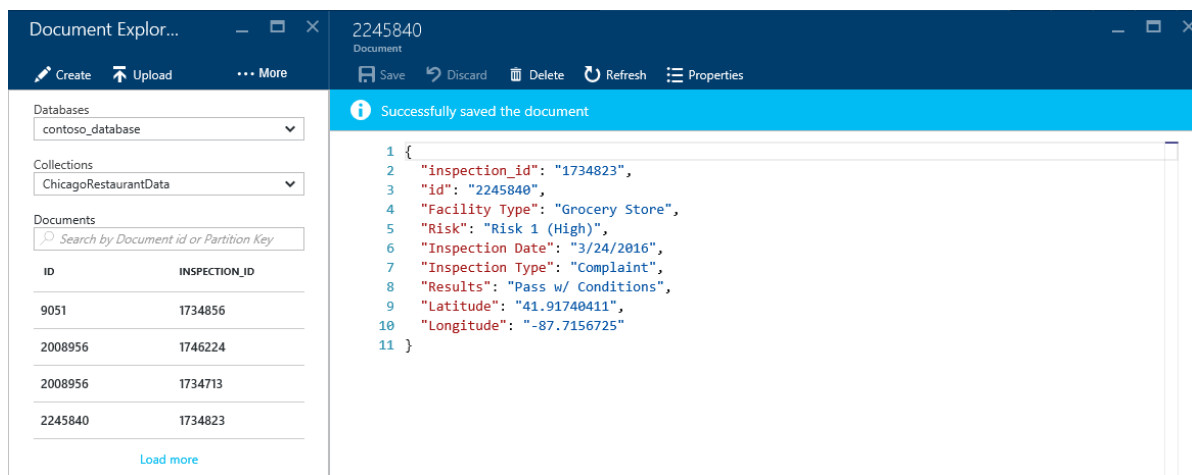
Create a document

1. [Launch Document Explorer](#).
2. In the **Document Explorer** blade, click **Create Document**.

A minimal JSON snippet is provided in the **Document** blade.



3. In the **Document** blade, type or paste in the content of the JSON document you wish to create, and then click **Save** to commit your document to the database and collection specified in the **Document Explorer** blade.



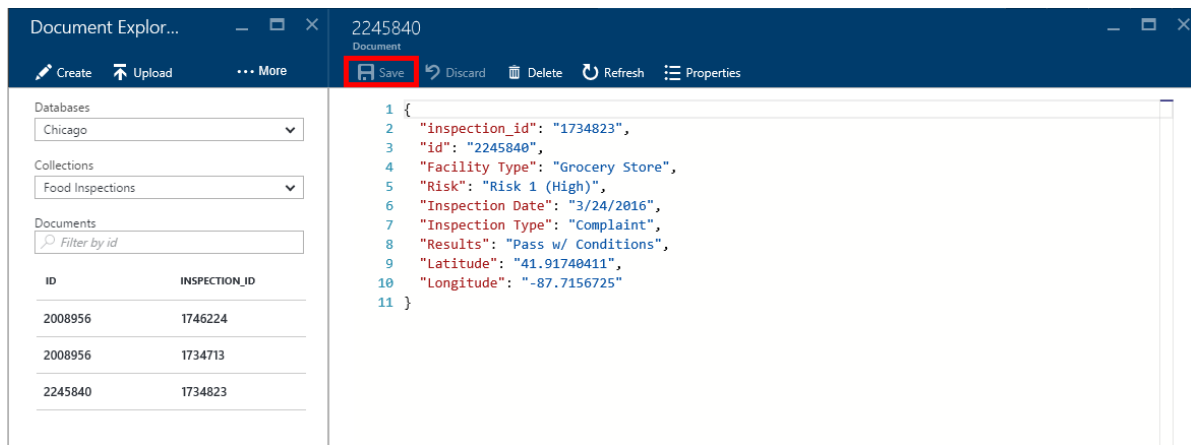
NOTE

If you do not provide an "id" property, then Document Explorer automatically adds an id property and generates a GUID as the id value.

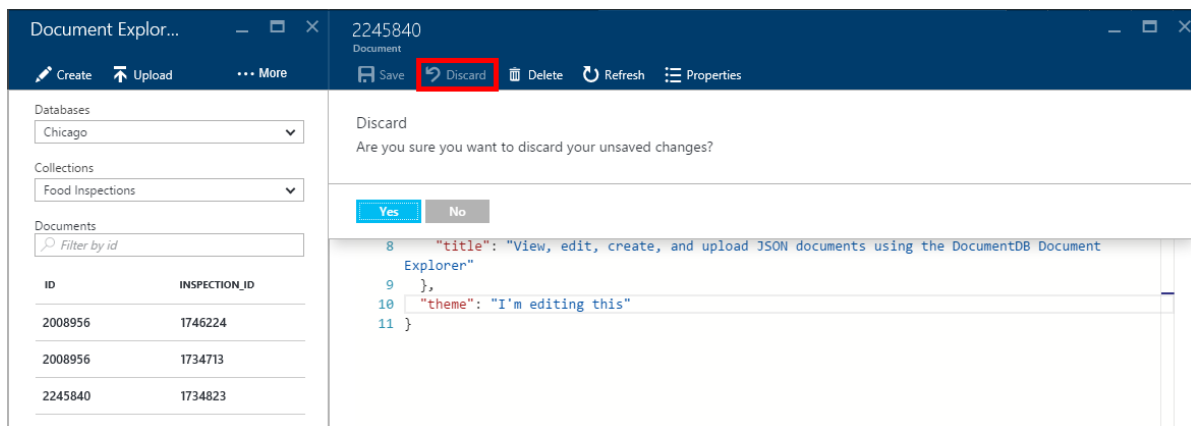
If you already have data from JSON files, MongoDB, SQL Server, CSV files, Azure Table storage, Amazon DynamoDB, HBase, or from other DocumentDB collections, you can use DocumentDB's [data migration tool](#) to quickly import your data.

Edit a document

1. [Launch Document Explorer](#).
2. To edit an existing document, select it in the **Document Explorer** blade, edit the document in the **Document** blade, and then click **Save**.

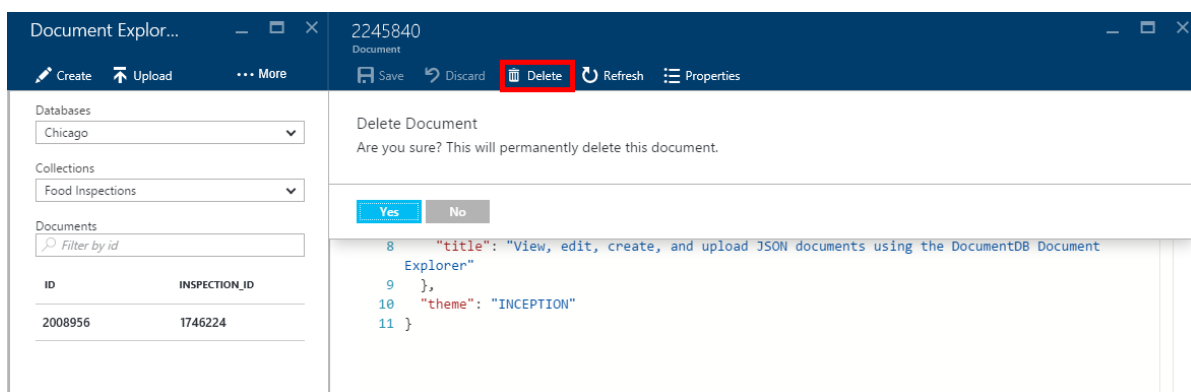


If you're editing a document and decide that you want to discard the current set of edits, simply click **Discard** in the **Document** blade, confirm the discard action, and the previous state of the document is reloaded.



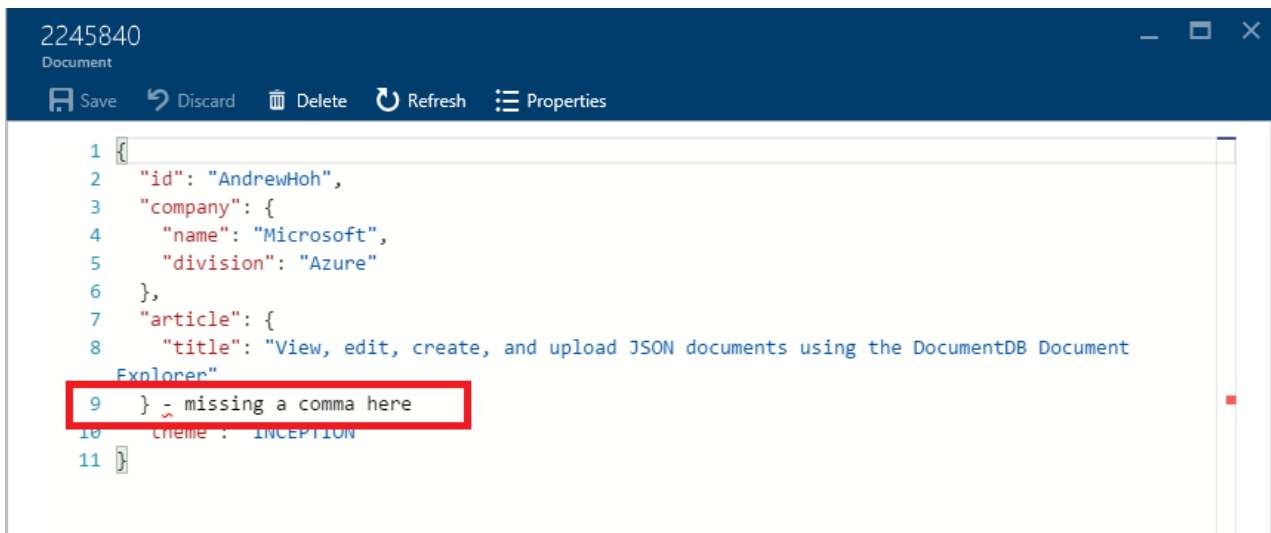
Delete a document

1. [Launch Document Explorer](#).
2. Select the document in **Document Explorer**, click **Delete**, and then confirm the delete. After confirming, the document is immediately removed from the Document Explorer list.

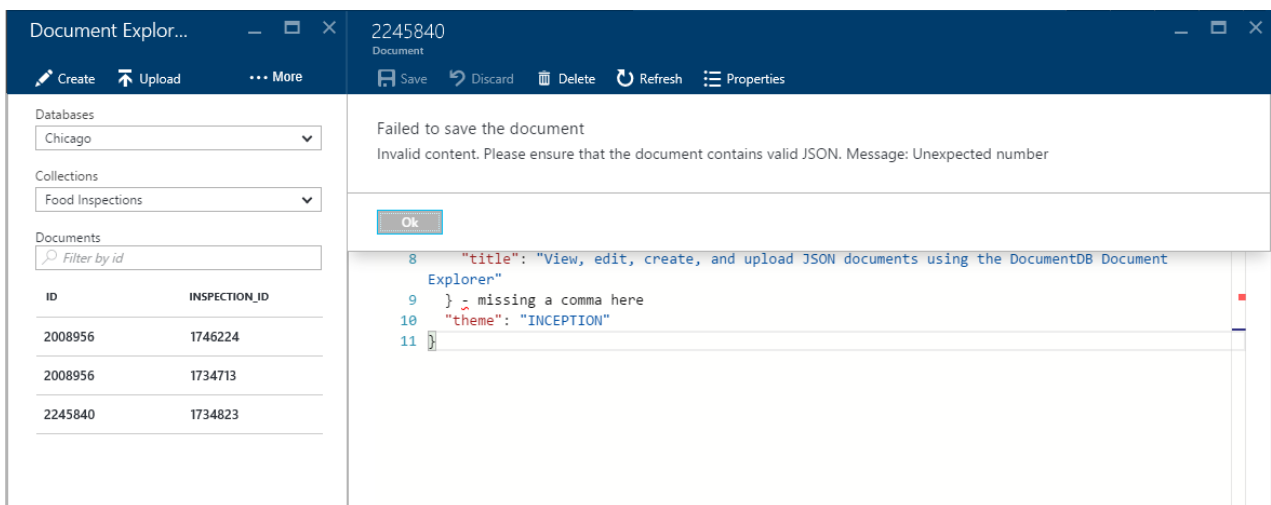


Work with JSON documents

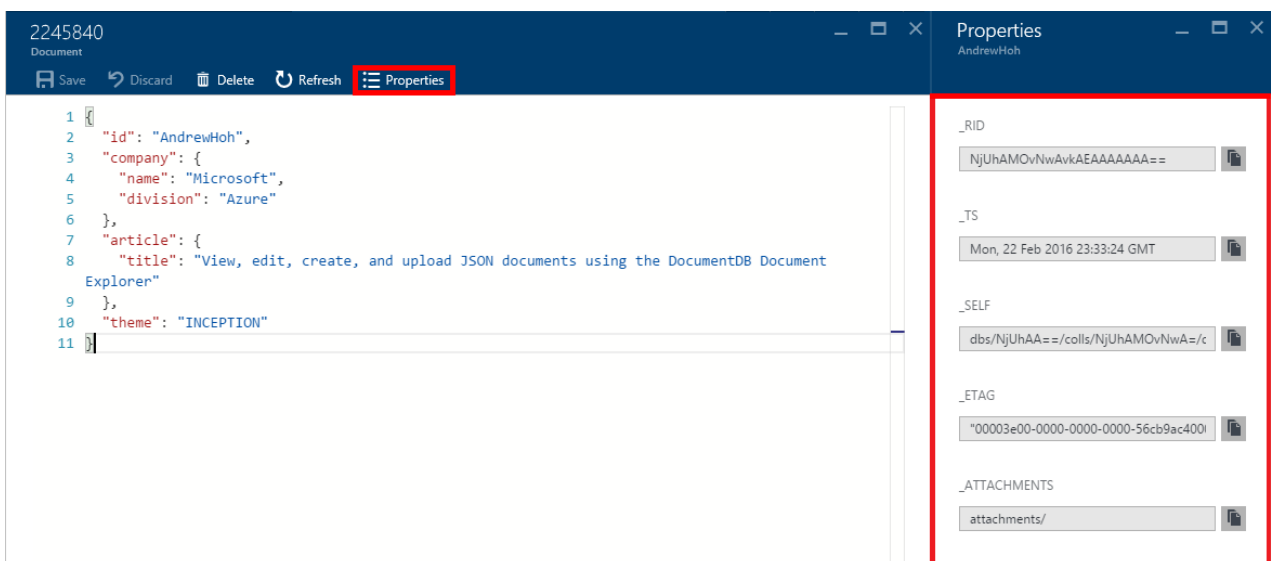
Document Explorer validates that any new or edited document contains valid JSON. You can even view JSON errors by hovering over the incorrect section to get details about the validation error.



Additionally, Document Explorer prevents you from saving a document with invalid JSON content.



Finally, Document Explorer allows you to easily view the system properties of the currently loaded document by clicking the **Properties** command.



NOTE

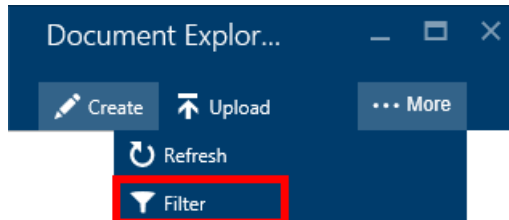
The timestamp (**_ts**) property is internally represented as epoch time, but Document Explorer displays the value in a human readable GMT format.

Filter documents

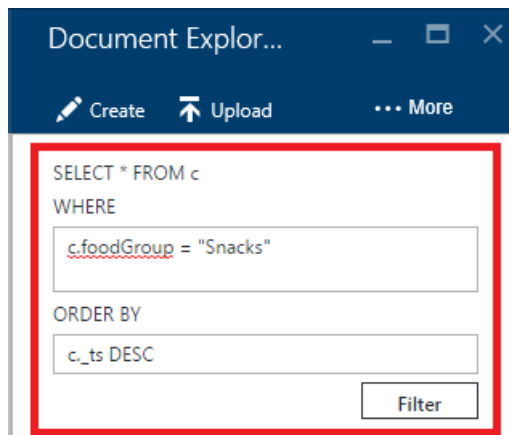
Document Explorer supports a number of navigation options and advanced settings.

By default, Document Explorer loads up to the first 100 documents in the selected collection, by their created date from earliest to latest. You can load additional documents (in batches of 100) by selecting the **Load more** option at the bottom of the Document Explorer blade. You can choose which documents to load through the **Filter** command.

1. [Launch Document Explorer](#).
2. At the top of the **Document Explorer** blade, click **Filter**.



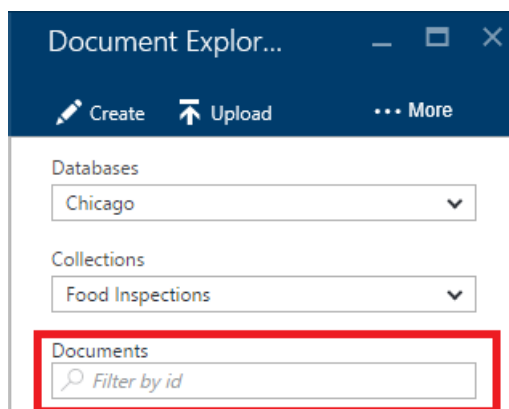
3. The filter settings appear below the command bar. In the filter settings, provide a WHERE clause and/or an ORDER BY clause, and then click **Filter**.



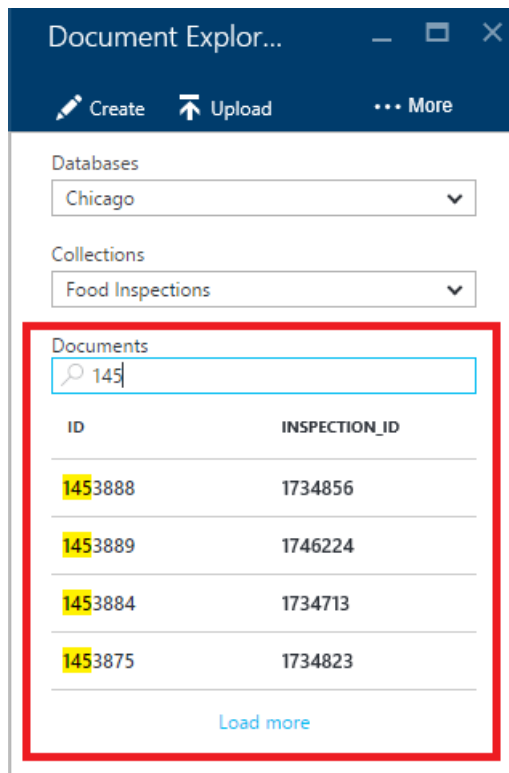
Document Explorer automatically refreshes the results with documents matching the filter query. Read more about the DocumentDB SQL grammar in the [SQL query and SQL syntax](#) article or print a copy of the [SQL query cheat sheet](#).

The **Database** and **Collection** drop-down list boxes can be used to easily change the collection from which documents are currently being viewed without having to close and re-launch Document Explorer.

Document Explorer also supports filtering the currently loaded set of documents by their id property. Simply type in the Documents Filter by id box.



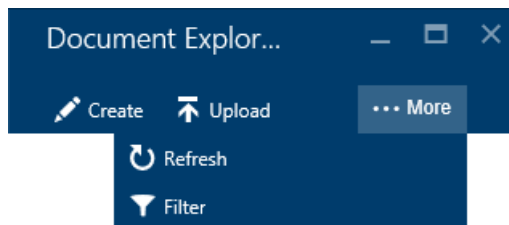
The results in the Document Explorer list are filtered based on your supplied criteria.



IMPORTANT

The Document Explorer filter functionality only filters from the *currently* loaded set of documents and does not perform a query against the currently selected collection.

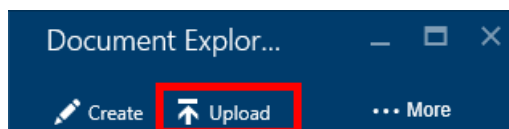
- To refresh the list of documents loaded by Document Explorer, click **Refresh** at the top of the blade.



Bulk add documents

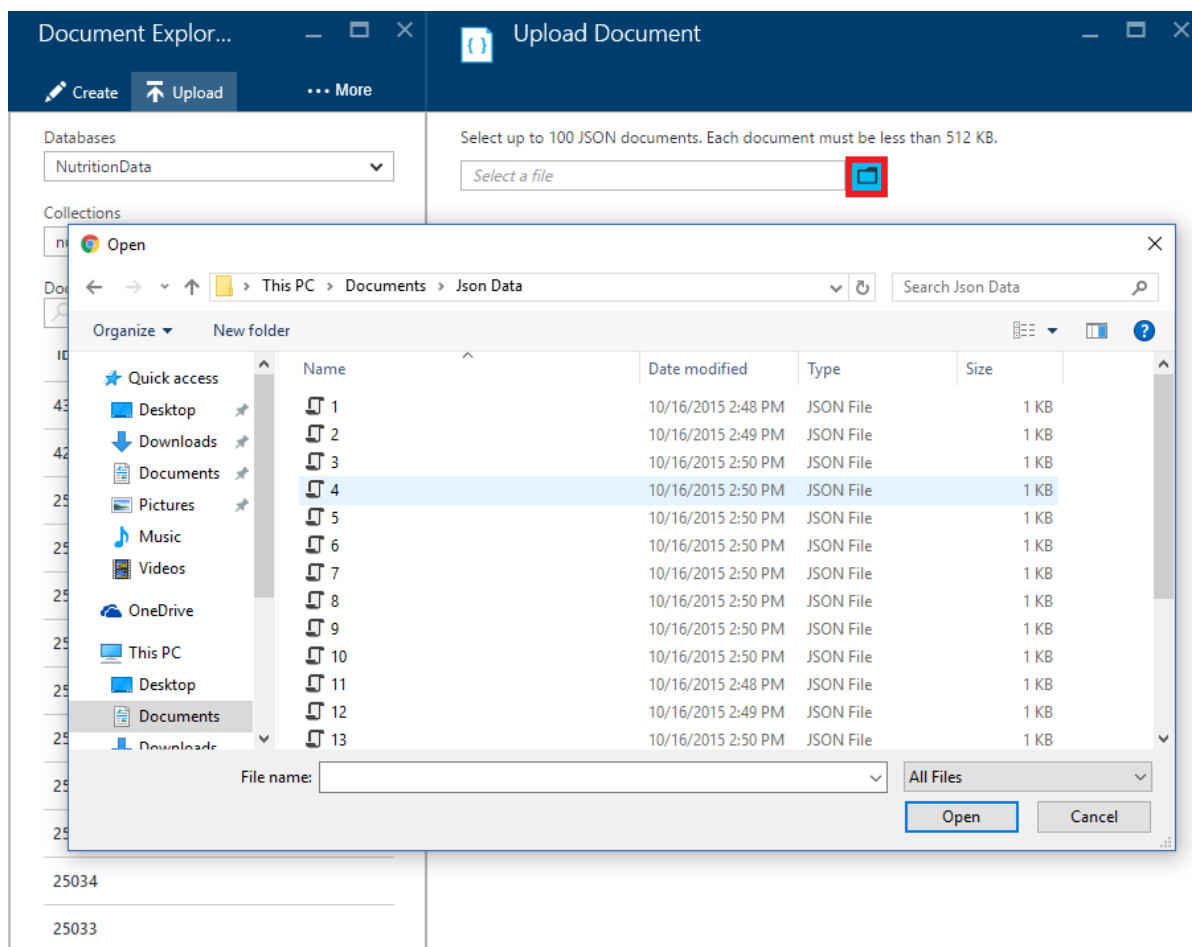
Document Explorer supports bulk ingestion of one or more existing JSON documents, up to 100 JSON files per upload operation.

- Launch Document Explorer.
- To start the upload process, click **Upload Document**.



The **Upload Document** blade opens.

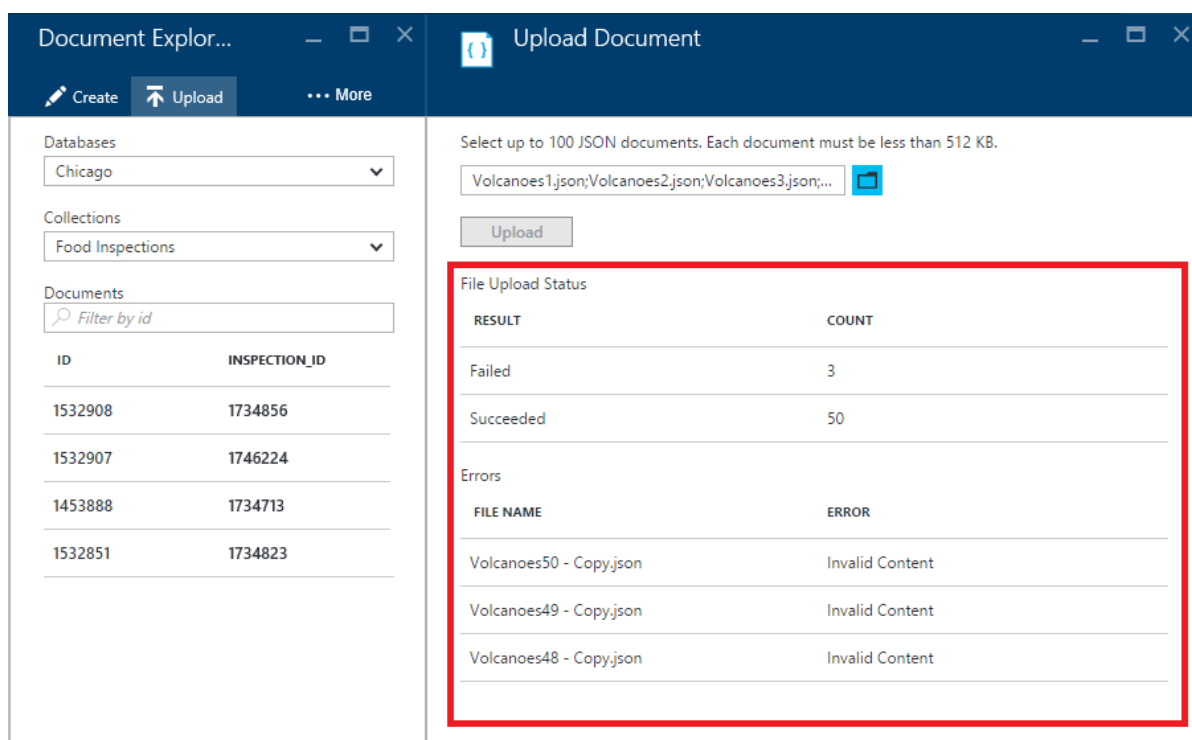
- Click the browse button to open a file explorer window, select one or more JSON documents to upload, and then click **Open**.



NOTE

Document Explorer currently supports up to 100 JSON documents per individual upload operation.

- Once you're satisfied with your selection, click the **Upload** button. The documents are automatically added to the Document Explorer grid and the upload results are displayed as the operation progresses. Import failures are reported for individual files.



5. Once the operation is complete, you can select up to another 100 documents to upload.

Work with JSON documents outside the portal

The Document Explorer in the Azure portal is just one way to work with documents in DocumentDB. You can also work with documents using the [REST API](#) or the [client SDKs](#). For example code, see the [.NET SDK document examples](#) and the [Node.js SDK document examples](#).

If you need to import or migrate files from another source (JSON files, MongoDB, SQL Server, CSV files, Azure Table storage, Amazon DynamoDB, or HBase), you can use the DocumentDB [data migration tool](#) to quickly import your data to DocumentDB.

Troubleshoot

Symptom: Document Explorer returns **No documents found**.

Solution: Ensure that you have selected the correct subscription, database and collection in which the documents were inserted. Also, check to ensure that you are operating within your throughput quotas. If you are operating at your maximum throughput level and getting throttled, lower application usage to operate under the maximum throughput quota for the collection.

Explanation: The portal is an application like any other, making calls to your DocumentDB database and collection. If your requests are currently being throttled due to calls being made from a separate application, the portal may also be throttled, causing resources not to appear in the portal. To resolve the issue, address the cause of the high throughput usage, and then refresh the portal blade. Information on how to measure and lower throughput usage can be found in the [Throughput](#) section of the [Performance tips](#) article.

Next steps

To learn more about the DocumentDB SQL grammar supported in Document Explorer, see the [SQL query and SQL syntax](#) article or print out the [SQL query cheat sheet](#).

The [Learning path](#) is also a useful resource to guide you as you learn more about DocumentDB.

Write, edit, and run SQL queries for DocumentDB using Query Explorer

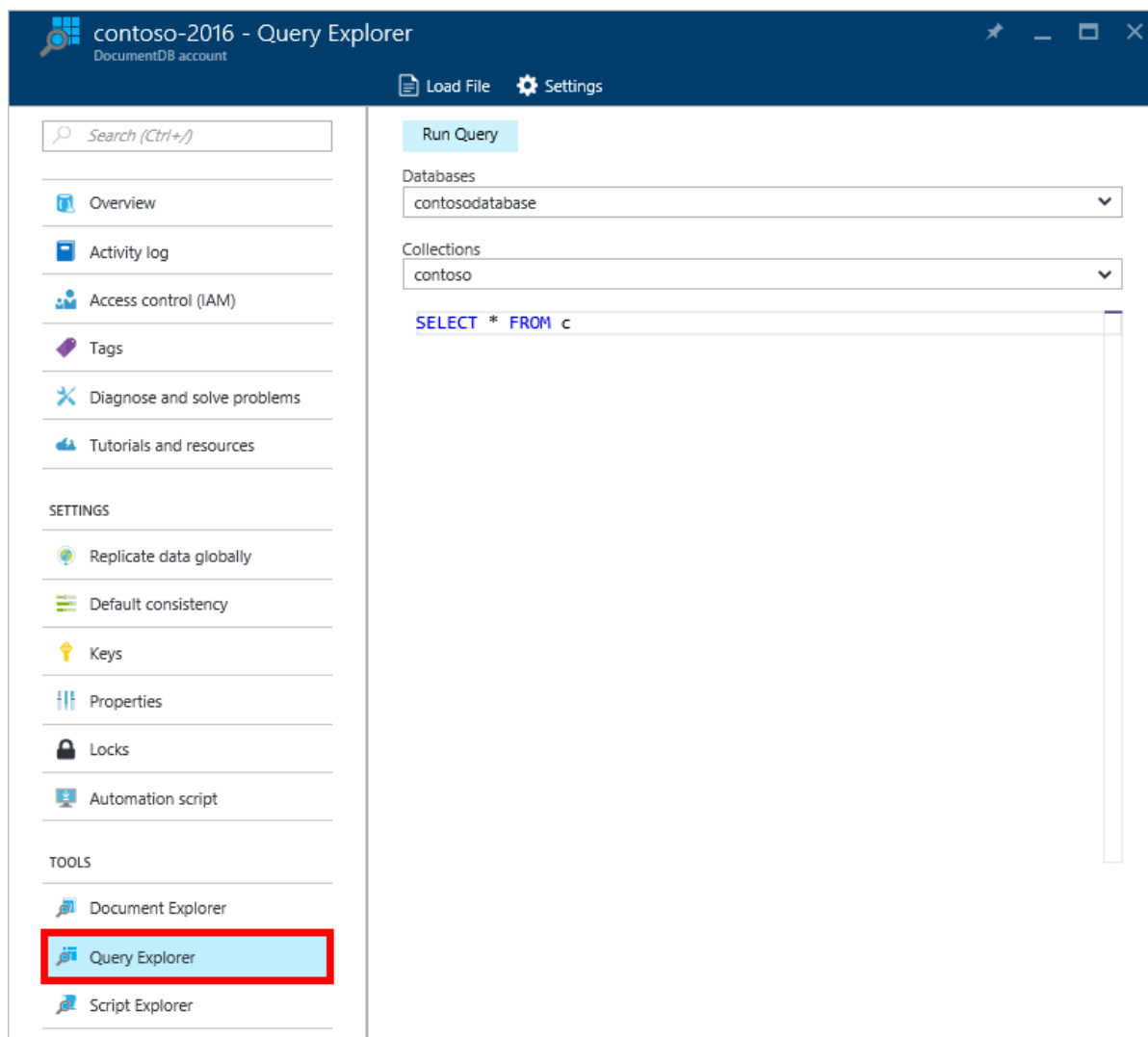
11/15/2016 • 2 min to read • [Edit on GitHub](#)

Contributors

Kirill Gavrylyuk • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • mimig • Andrew Hoh • v-aljenk • pcw3187 • Stephen Baron • Dene Hager

This article provides an overview of the [Microsoft Azure DocumentDB](#) Query Explorer, an Azure portal tool that enables you to write, edit, and run SQL queries against a [DocumentDB collection](#).

1. In the Azure portal, in the Jumpbar, click **DocumentDB (NoSQL)**. If **DocumentDB (NoSQL)** is not visible, click **More Services** and then click **DocumentDB (NoSQL)**.
2. In the resource menu, click **Query Explorer**.

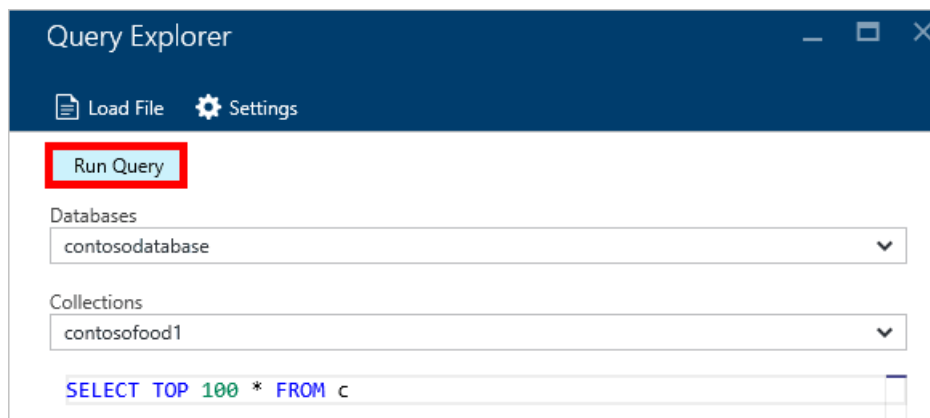


3. In the **Query Explorer** blade, select the **Databases** and **Collections** to query from the drop down lists, and type the query to run.

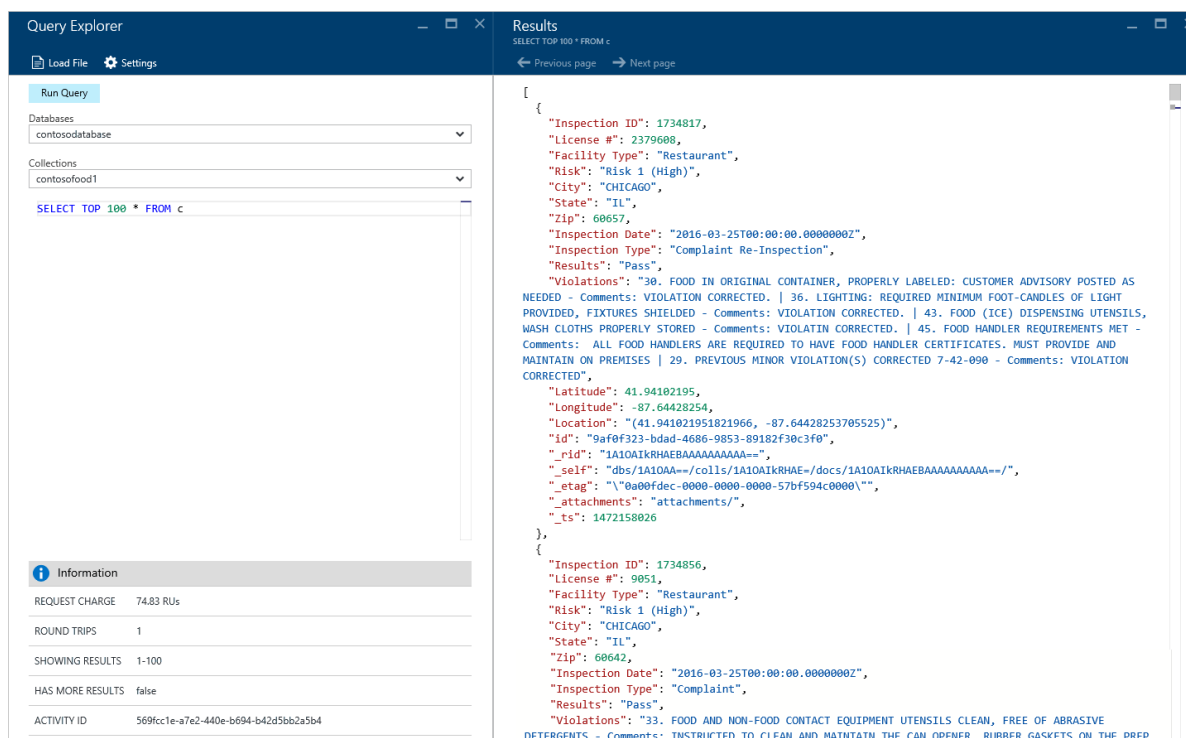
The **Databases** and **Collections** drop-down lists are pre-populated depending on the context in which you launch Query Explorer.

A default query of `SELECT TOP 100 * FROM c` is provided. You can accept the default query or construct your own query using the SQL query language described in the [SQL query cheat sheet](#) or the [SQL query and SQL syntax](#) article.

Click **Run query** to view the results.



4. The **Results** blade displays the output of the query.



Work with results

By default, Query Explorer returns results in sets of 100. If your query produces more than 100 results, simply use the **Next page** and **Previous page** commands to navigate through the result set.



For successful queries, the **Information** pane contains metrics such as the request charge, the number of round trips the query made, the set of results currently being shown, and whether there are more results, which can then be accessed via the **Next page** command, as mentioned previously.

Query Explorer

Load File

Settings

Run Query

Databases

contosodatabase

Collections

contosofood1

SELECT TOP 100 * FROM c

Information

REQUEST CHARGE74.83 RUs

ROUND TRIPS1

SHOWING RESULTS1-100

HAS MORE RESULTSfalse

ACTIVITY ID569fcc1e-a7e2-440e-b694-b42d5bb2a5b4

Use multiple queries

If you're using multiple queries and want to quickly switch between them, you can enter all the queries in the query text box of the **Query Explorer** blade, then highlight the one you want to run, and then click **Run query** to view the results.

Query Explorer

Load File

Settings

Run query

Databases

NutritionData

Collections

nutrition1

--Return the top 100 results from collection
SELECT TOP 100 * FROM c

--Return all documents with the foodGroup "Baby Foods"
SELECT * FROM c WHERE c.foodGroup = "Baby Foods"

--Return all nutrients from document 09052 with a nutrition value greater than 5mg
SELECT nutrient.id,
 nutrient.description,
 nutrient.nutritionValue,
 nutrient.units
FROM food
JOIN nutrient IN food.nutrients
WHERE nutrient.nutritionValue > 5 AND nutrient.units = "mg" AND
 food.id = "09052"

Information

REQUEST CHARGE

2.8 RU\$

ROUND TRIPS

1

SHOWING RESULTS

1-2

HAS MORE RESULTS

false

ACTIVITY ID

f4c19ad5-ddbd-43ba-a97f-286dcb1a4bbe

Results

Previous page

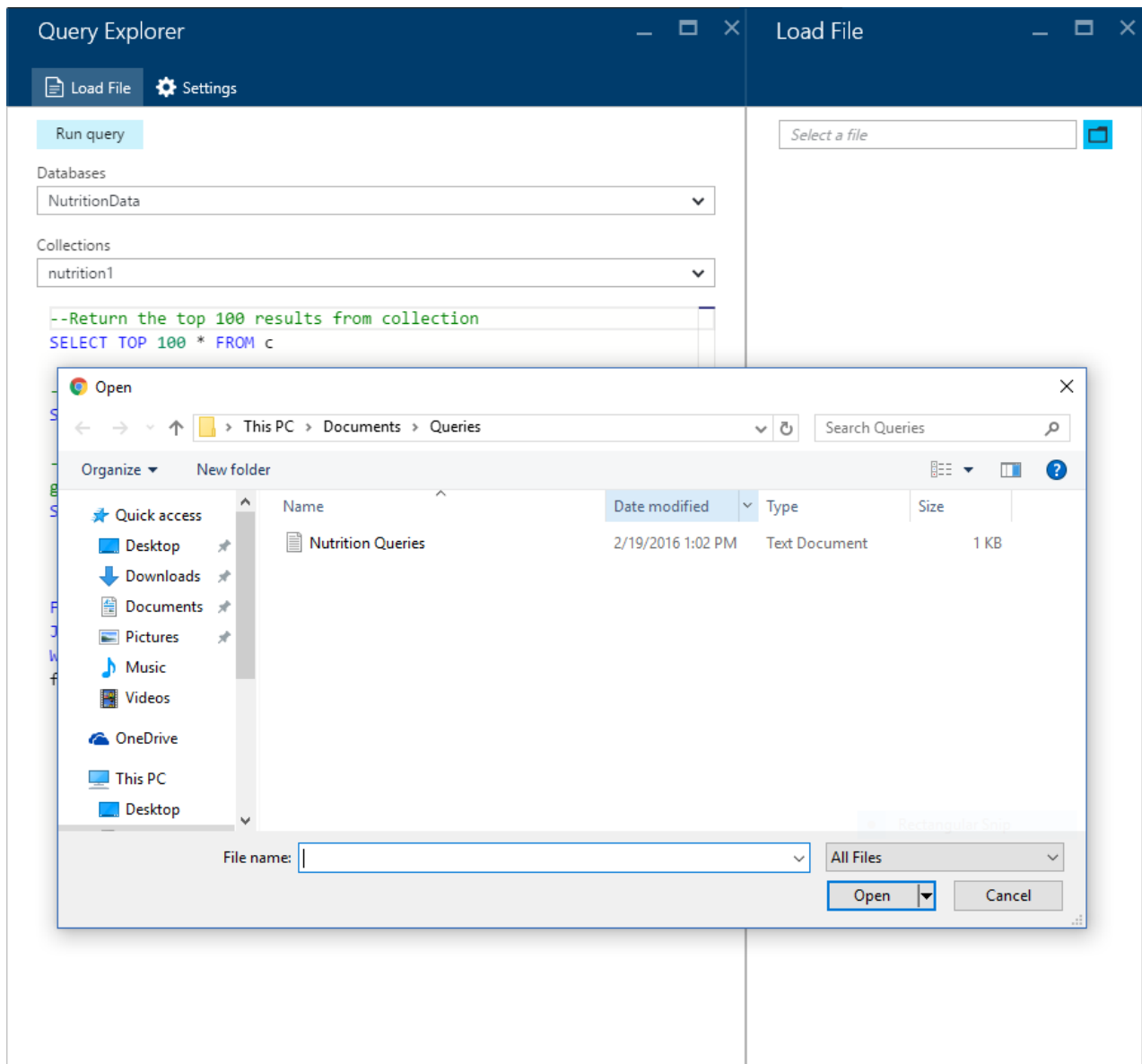
Next page

```
[
  {
    "id": "305",
    "description": "Phosphorus, P",
    "nutritionValue": 10,
    "units": "mg"
  },
  {
    "id": "306",
    "description": "Potassium, K",
    "nutritionValue": 40,
    "units": "mg"
  }
]
```

Rectangular Snip

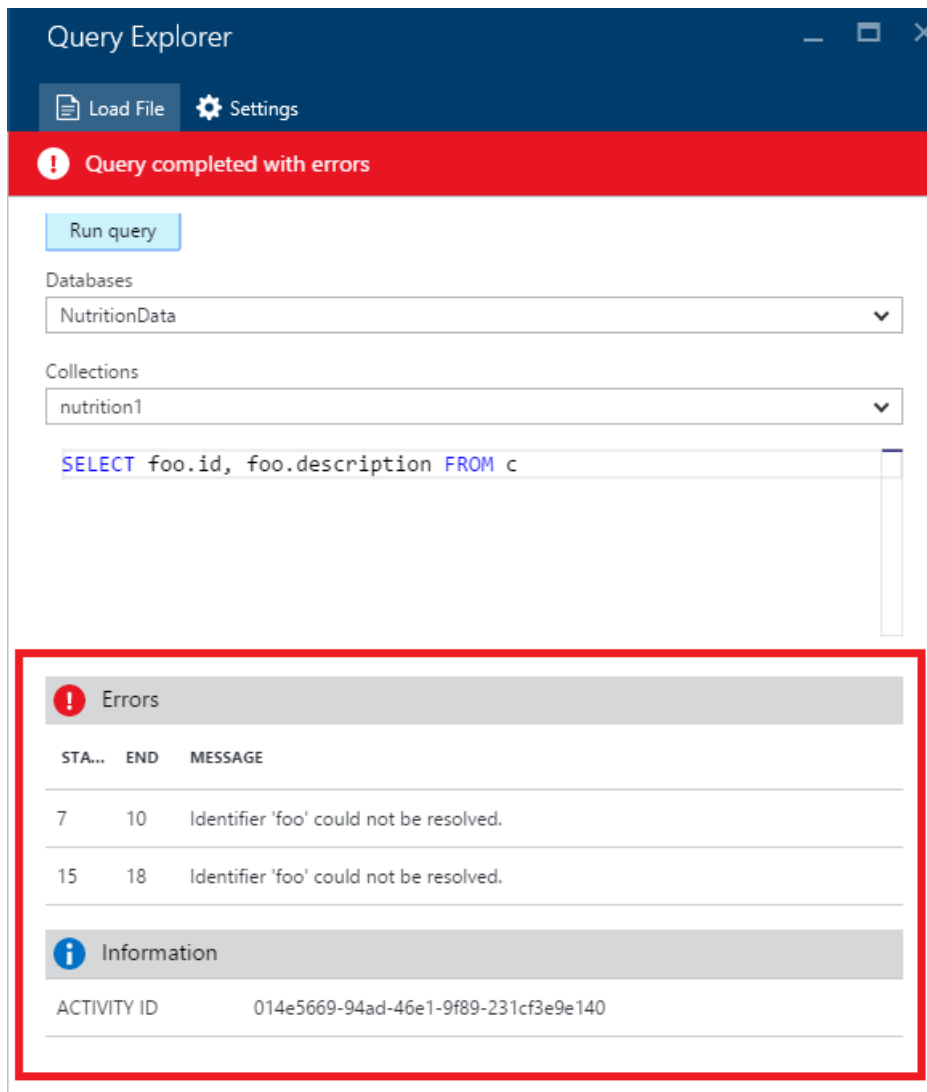
Add queries from a file into the SQL query editor

You can load the contents of an existing file using the **Load File** command.



Troubleshoot

If a query completes with errors, Query Explorer displays a list of errors that can help with troubleshooting efforts.



Run DocumentDB SQL queries outside the portal

The Query Explorer in the Azure portal is just one way to run SQL queries against DocumentDB. You can also run SQL queries using the [REST API](#) or the [client SDKs](#). For more information about using these other methods, see [Executing SQL queries](#)

Next steps

To learn more about the DocumentDB SQL grammar supported in Query Explorer, see the [SQL query and SQL syntax](#) article or print out the [SQL query cheat sheet](#). You may also enjoy experimenting with the [Query Playground](#) where you can test out queries online using a sample dataset.

How to manage a DocumentDB account

11/15/2016 • 3 min to read • [Edit on GitHub](#)

Contributors

Kirill Gavrylyuk • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • mimig • Armando Trejo Oliver • Andrew Hoh • Kirat Pandya • v-aljenk • Jennifer Hubbard • Stephen Baron

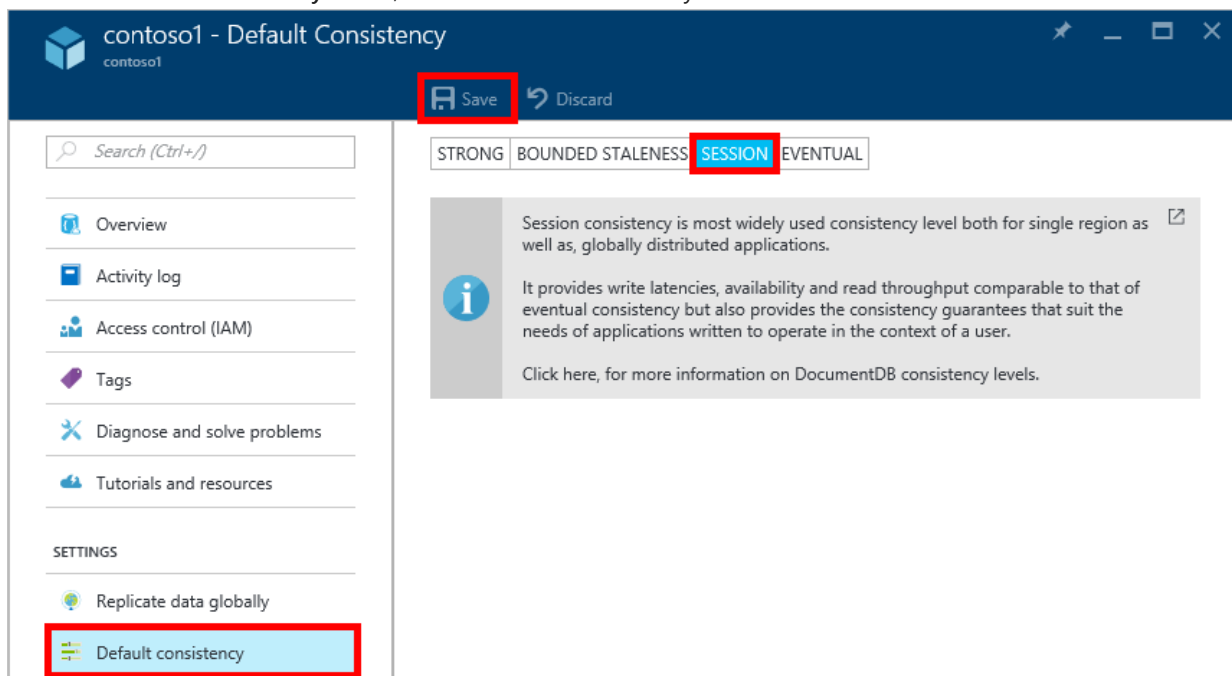
Learn how to set global consistency, work with keys, and delete a DocumentDB account in the Azure portal.

Manage DocumentDB consistency settings

Selecting the right consistency level depends on the semantics of your application. You should familiarize yourself with the available consistency levels in DocumentDB by reading [Using consistency levels to maximize availability and performance in DocumentDB](#). DocumentDB provides consistency, availability, and performance guarantees, at every consistency level available for your database account. Configuring your database account with a consistency level of Strong requires that your data is confined to a single Azure region and not globally available. On the other hand, the relaxed consistency levels - bounded staleness, session or eventual enable you to associate any number of Azure regions with your database account. The following simple steps show you how to select the default consistency level for your database account.

To specify the default consistency for a DocumentDB account

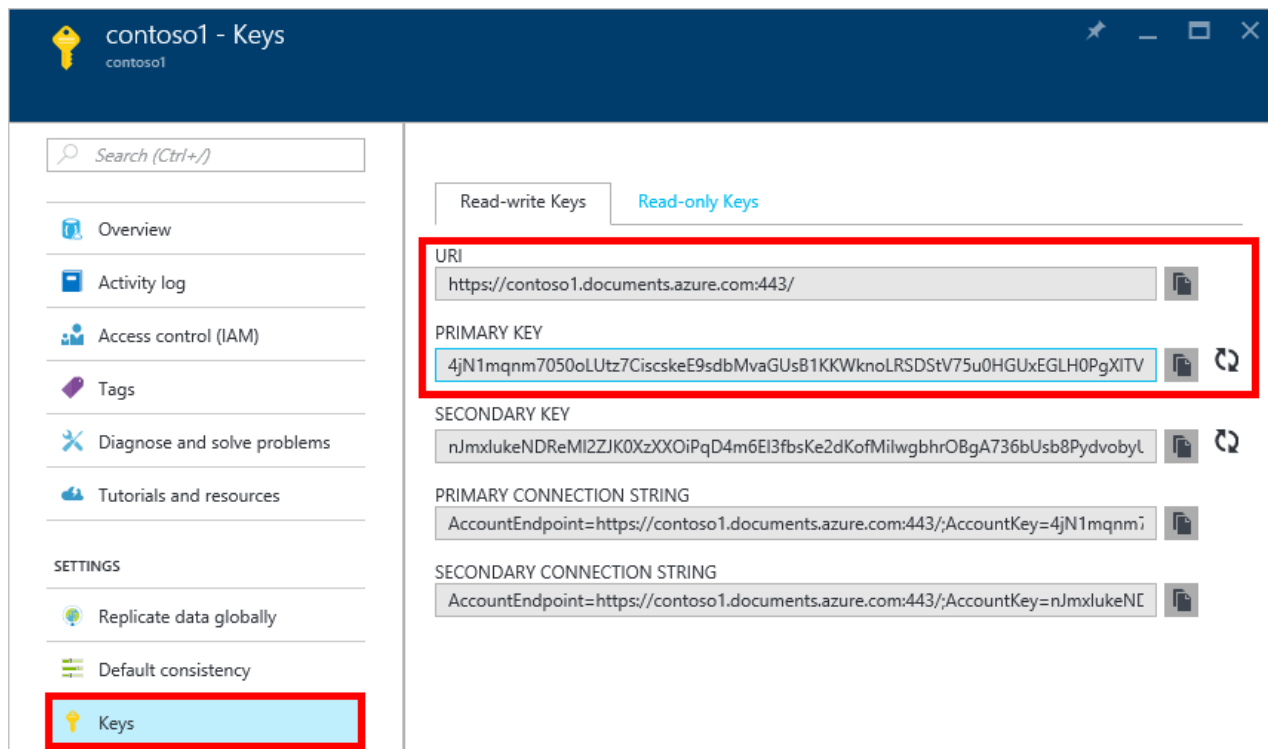
1. In the [Azure portal](#), access your DocumentDB account.
2. In the account blade, click **Default consistency**.
3. In the **Default Consistency** blade, select the new consistency level and click **Save**.



View, copy, and regenerate access keys

When you create a DocumentDB account, the service generates two master access keys that can be used for authentication when the DocumentDB account is accessed. By providing two access keys, DocumentDB enables you to regenerate the keys with no interruption to your DocumentDB account.

In the [Azure portal](#), access the **Keys** blade from the resource menu on the **DocumentDB account** blade to view, copy, and regenerate the access keys that are used to access your DocumentDB account.



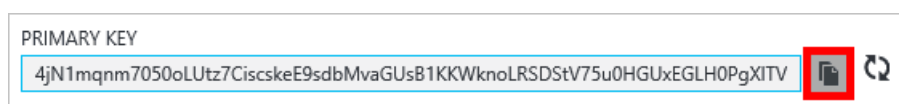
NOTE

The **Keys** blade also includes primary and secondary connection strings that can be used to connect to your account from the [Data Migration Tool](#).

Read-only keys are also available on this blade. Reads and queries are read-only operations, while creates, deletes, and replaces are not.

Copy an access key in the Azure Portal

On the **Keys** blade, click the **Copy** button to the right of the key you wish to copy.



Regenerate access keys

You should change the access keys to your DocumentDB account periodically to help keep your connections more secure. Two access keys are assigned to enable you to maintain connections to the DocumentDB account using one access key while you regenerate the other access key.

WARNING

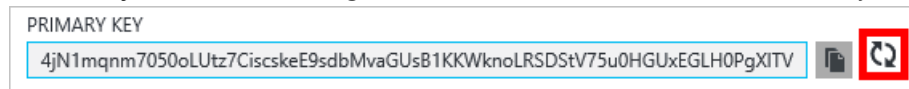
Regenerating your access keys affects any applications that are dependent on the current key. All clients that use the access key to access the DocumentDB account must be updated to use the new key.

If you have applications or cloud services using the DocumentDB account, you will lose the connections if you regenerate keys, unless you roll your keys. The following steps outline the process involved in rolling your keys.

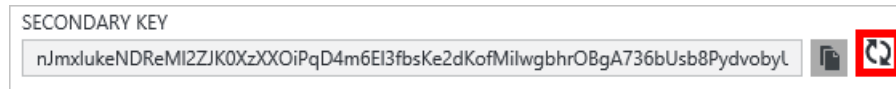
1. Update the access key in your application code to reference the secondary access key of the DocumentDB account.
2. Regenerate the primary access key for your DocumentDB account. In the [Azure Portal](#), access your

DocumentDB account.

3. In the **DocumentDB Account** blade, click **Keys**.
4. On the **Keys** blade, click the regenerate button, then click **Ok** to confirm that you want to generate a new key.



5. Once you have verified that the new key is available for use (approximately 5 minutes after regeneration), update the access key in your application code to reference the new primary access key.
6. Regenerate the secondary access key.



NOTE

It can take several minutes before a newly generated key can be used to access your DocumentDB account.

Get the connection string

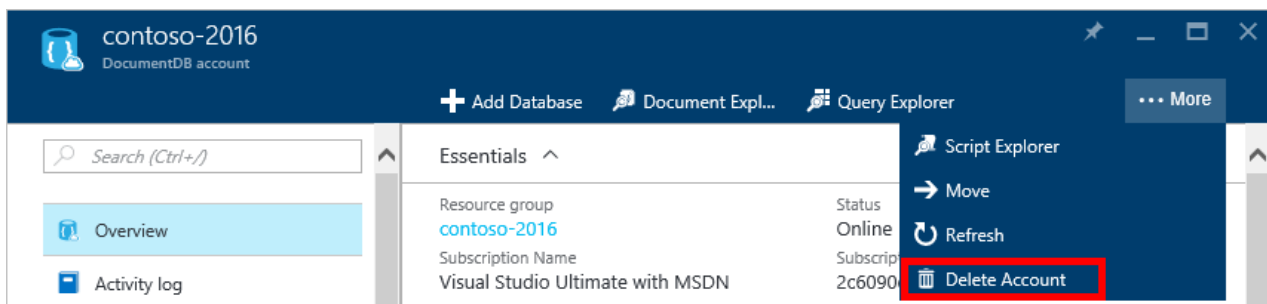
To retrieve your connection string, do the following:

1. In the [Azure portal](#), access your DocumentDB account.
2. In the resource menu, click **Keys**.
3. Click the **Copy** button next to the **Primary Connection String** or **Secondary Connection String** box.

If you are using the connection string in the [DocumentDB Database Migration Tool](#), append the database name to the end of the connection string. `AccountEndpoint=< >;AccountKey=< >;Database=< > .`


Delete a DocumentDB account

To remove a DocumentDB account from the Azure Portal that you are no longer using, use the **Delete Account** command on the **DocumentDB account** blade.



1. In the [Azure portal](#), access the DocumentDB account you wish to delete.
2. On the **DocumentDB account** blade, click **More**, and then click **Delete Account**. Or, right-click the name of the database, and click **Delete Account**.
3. On the resulting confirmation blade, type the DocumentDB account name to confirm that you want to delete the account.
4. Click the **Delete** button.

Are you sure you want to delete contoso-2016?



Warning! Deleting contoso-2016 is irreversible. The action you're about to take can't be undone. Going further will delete it and all the items in it permanently.

TYPE THE DOCUMENTDB (NOSQL) NAME

contoso-2016

✕

✓

Affected items

	ID	STATUS	LOCATION	SUBSCRIPTION
✓	contoso-2016	Online	West US	Visual Studio Ultima...

Delete

Cancel

Next steps

Learn how to [get started with your DocumentDB account](#).

To learn more about DocumentDB, see the Azure DocumentDB documentation on [azure.com](#).

Monitor DocumentDB requests, usage, and storage

11/15/2016 • 4 min to read • [Edit on GitHub](#)

Contributors

mimig • [Kim Whitlatch](#) (Beyondsoft Corporation) • [Tyson Nevil](#) • [Rob Boucher](#) • [v-aljenk](#) • [Jennifer Hubbard](#)

You can monitor your Azure DocumentDB accounts in the [Azure portal](#). For each DocumentDB account, both performance metrics, such as requests and server errors, and usage metrics, such as storage consumption, are available.

Metrics can be reviewed on the Account blade or on the new Metrics blade.

View performance metrics on the Metrics blade

1. In a new window, open the [Azure portal](#), click **More Services**, click **DocumentDB (NoSQL)**, and then click the name of the DocumentDB account for which you would like to view performance metrics.
2. In the resource menu, click **Metrics**.

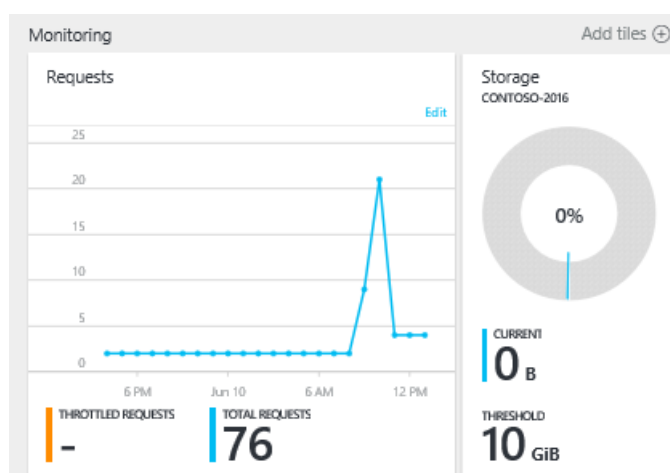
The Metrics blade opens, and you can select the collection to review. You can review Availability, Requests, Throughput, and Storage metrics and compare them to the DocumentDB SLAs.

View performance metrics on the account blade

1. In a new window, open the [Azure portal](#), click **More Services**, click **DocumentDB (NoSQL)**, and then click the name of the DocumentDB account for which you would like to view performance metrics.
2. The **Monitoring** lens displays the following tiles by default:

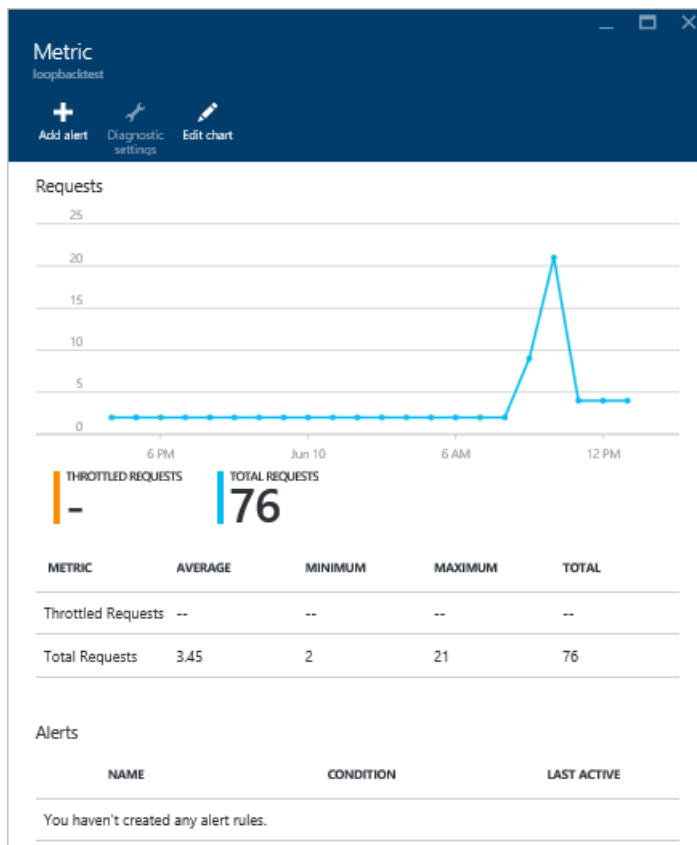
- Total requests for the current day.
- Storage used.

If your table displays **No data available** and you believe there is data in your database, see the [Troubleshooting](#) section.



3. Clicking on the **Requests** or **Storage** tile opens a detailed **Metric** blade.
4. The **Metric** blade shows you details about the metrics you have selected. At the top of the blade is a graph of requests charted hourly, and below that is a table that shows aggregation values for throttled and total requests. The metric blade also shows the list of alerts which have been defined, filtered to the metrics that appear on the current metric blade (this way, if you have a number of alerts, you'll only see the relevant

ones presented here).



Customize performance metric views in the portal

1. To customize the metrics that display in a particular chart, click the chart to open it in the **Metric** blade, and then click **Edit chart**.



2. On the **Edit Chart** blade, there are options to modify the metrics that display in the chart, as well as their time range.

Edit Chart

Time Range

past hour today past week custom

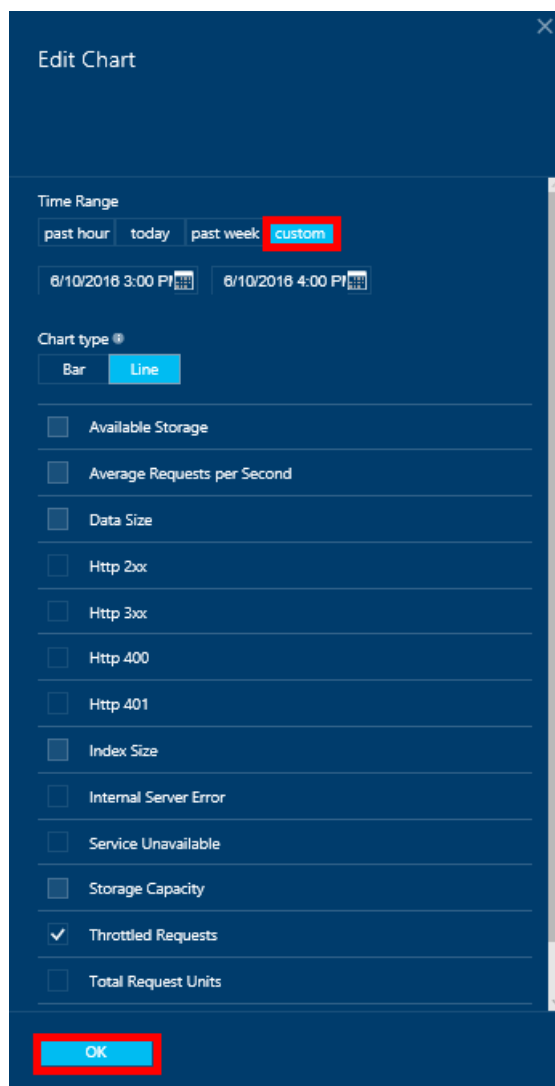
Chart type

Bar Line

- ☐ Available Storage
- ☐ Average Requests per Second
- ☐ Data Size
- ☐ Http 2xx
- ☐ Http 3xx
- ☐ Http 400
- ☐ Http 401
- ☐ Index Size
- ☐ Internal Server Error
- ☐ Service Unavailable
- ☐ Storage Capacity
- ☒ Throttled Requests
- ☐ Total Request Units
- ☒ Total Requests

OK

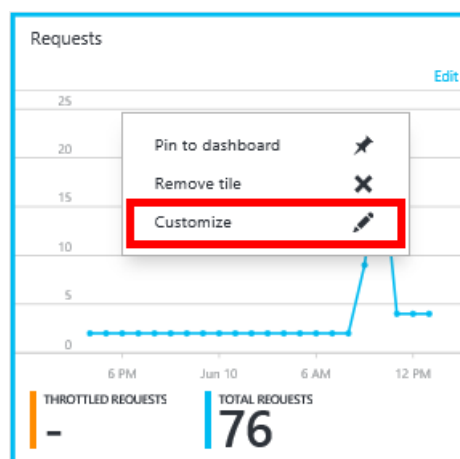
3. To change the metrics displayed in the part, simply select or clear the available performance metrics, and then click **OK** at the bottom of the blade.
4. To change the time range, choose a different range (for example, **Custom**), and then click **OK** at the bottom of the blade.



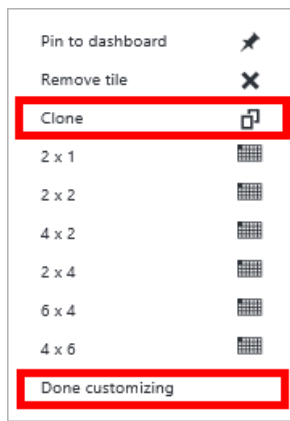
Create side-by-side charts in the portal

The Azure Portal allows you to create side-by-side metric charts.

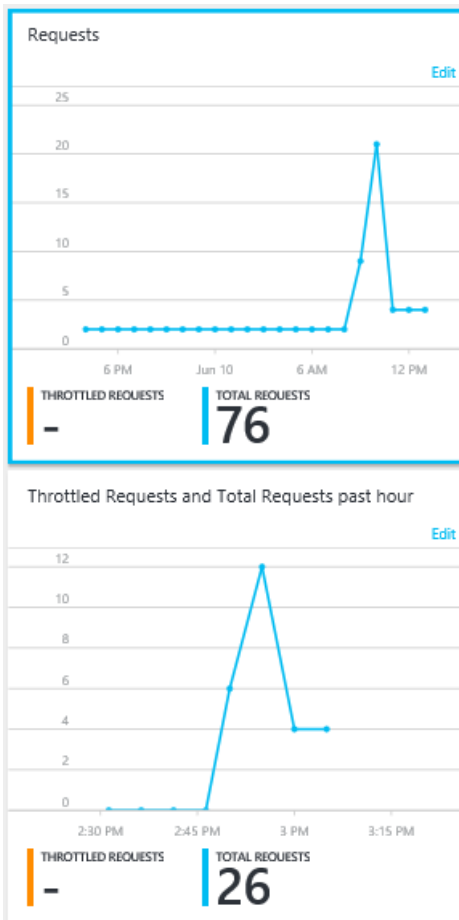
1. First, right-click on the chart you want to copy and select **Customize**.



2. Click **Clone** on the menu to copy the part and then click **Done customizing**.

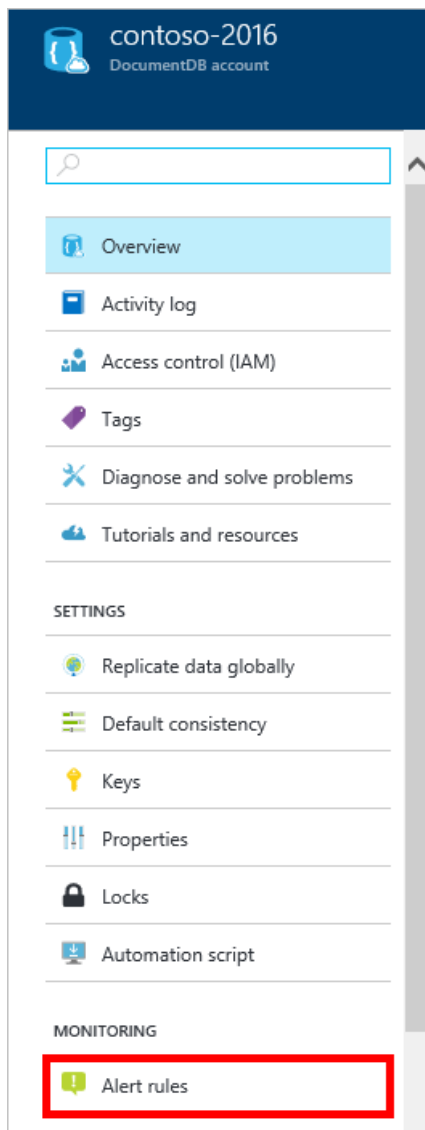


You may now treat this part as any other metric part, customizing the metrics and time range displayed in the part. By doing this, you can see two different metrics chart side-by-side at the same time.

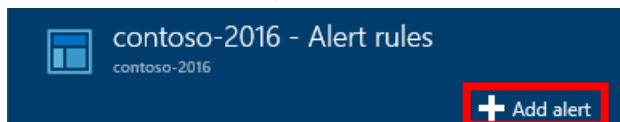


Set up alerts in the portal

1. In the [Azure portal](#), click **More Services**, click **DocumentDB (NoSQL)**, and then click the name of the DocumentDB account for which you would like to setup performance metric alerts.
2. In the resource menu, click **Alert Rules** to open the Alert rules blade.



3. In the **Alert rules** blade, click **Add alert**.



4. In the **Add an alert rule** blade, specify:

- The name of the alert rule you are setting up.
- A description of the new alert rule.
- The metric for the alert rule.
- The condition, threshold, and period that determine when the alert activates. For example, a server error count greater than 5 over the last 15 minutes.
- Whether the service administrator and coadministrators are emailed when the alert fires.
- Additional email addresses for alert notifications.

Add an alert rule

* Resource ⓘ

contoso-2016 (databaseAccounts)

* Name ⓘ

Name

Description

Description

* Metric ⓘ

Available Storage

Not enough data to chart yet.

* Condition

greater than

* Threshold ⓘ

1

bytes

* Period ⓘ

Over the last 30 minutes

Email owners, contributors, and readers

☐

Additional administrator email(s)

Add email addresses separated by semicolons

OK

Monitor DocumentDB programmatically

The account level metrics available in the portal, such as account storage usage and total requests, are not available via the DocumentDB APIs. However, you can retrieve usage data at the collection level by using the DocumentDB APIs. To retrieve collection level data, do the following:

- To use the REST API, [perform a GET on the collection](#). The quota and usage information for the collection is returned in the x-ms-resource-quota and x-ms-resource-usage headers in the response.
- To use the .NET SDK, use the [DocumentClient.ReadDocumentCollectionAsync](#) method, which returns a [ResourceResponse](#) that contains a number of usage properties such as **CollectionSizeUsage**, **DatabaseUsage**, **DocumentUsage**, and more.

To access additional metrics, use the [Azure Monitor SDK](#). Available metric definitions can be retrieved by calling:

```
https://management.azure.com/subscriptions/{SubscriptionId}/resourceGroups/{ResourceGroup}/providers/Microsoft.DocumentDb/databaseAccounts/{DocumentDBAccountName}/metricDefinitions?api-version=2015-04-08
```

Queries to retrieve individual metrics use the following format:

```
https://management.azure.com/subscriptions/{SubscriptionId}/resourceGroups/{ResourceGroup}/providers/Microsoft.DocumentDb/databaseAccounts/{DocumentDBAccountName}/metrics?api-version=2015-04-08&$filter=%28name.value%20eq%20%27Total%20Requests%27%29%20and%20timeGrain%20eq%20duration%27PT5M%27%20and%20startTime%20eq%202016-06-03T03%3A26%3A00.0000000Z%20and%20endTime%20eq%202016-06-10T03%3A26%3A00.0000000Z
```

For more information, see [Retrieving Resource Metrics via the Azure Monitor REST API](#). Note that "Azure Insights" was renamed "Azure Monitor". This blog entry refers to the older name.

Troubleshooting

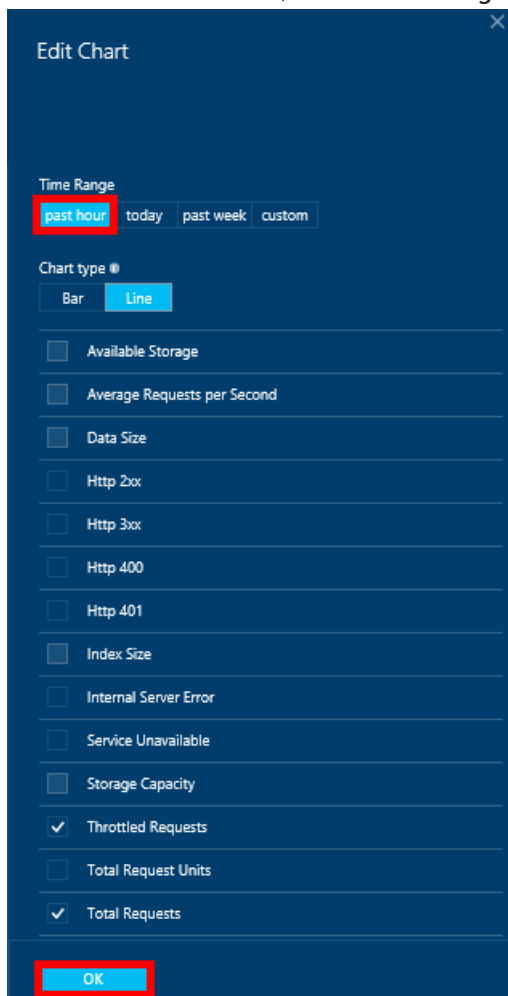
If your monitoring tiles display the **No data available** message, and you recently made requests or added data to the database, you can edit the tile to reflect the recent usage.

Edit a tile to refresh current data

1. To customize the metrics that display in a particular part, click the chart to open the **Metric** blade, and then click **Edit Chart**.



2. On the **Edit Chart** blade, in the **Time Range** section, click **past hour**, and then click **OK**.



3. Your tile should now refresh showing your current data and usage.



Next steps

To learn more about DocumentDB capacity, see [Manage DocumentDB capacity](#).

Create and run stored procedures, triggers, and user-defined functions using the DocumentDB Script Explorer

11/15/2016 • 3 min to read • [Edit on GitHub](#)

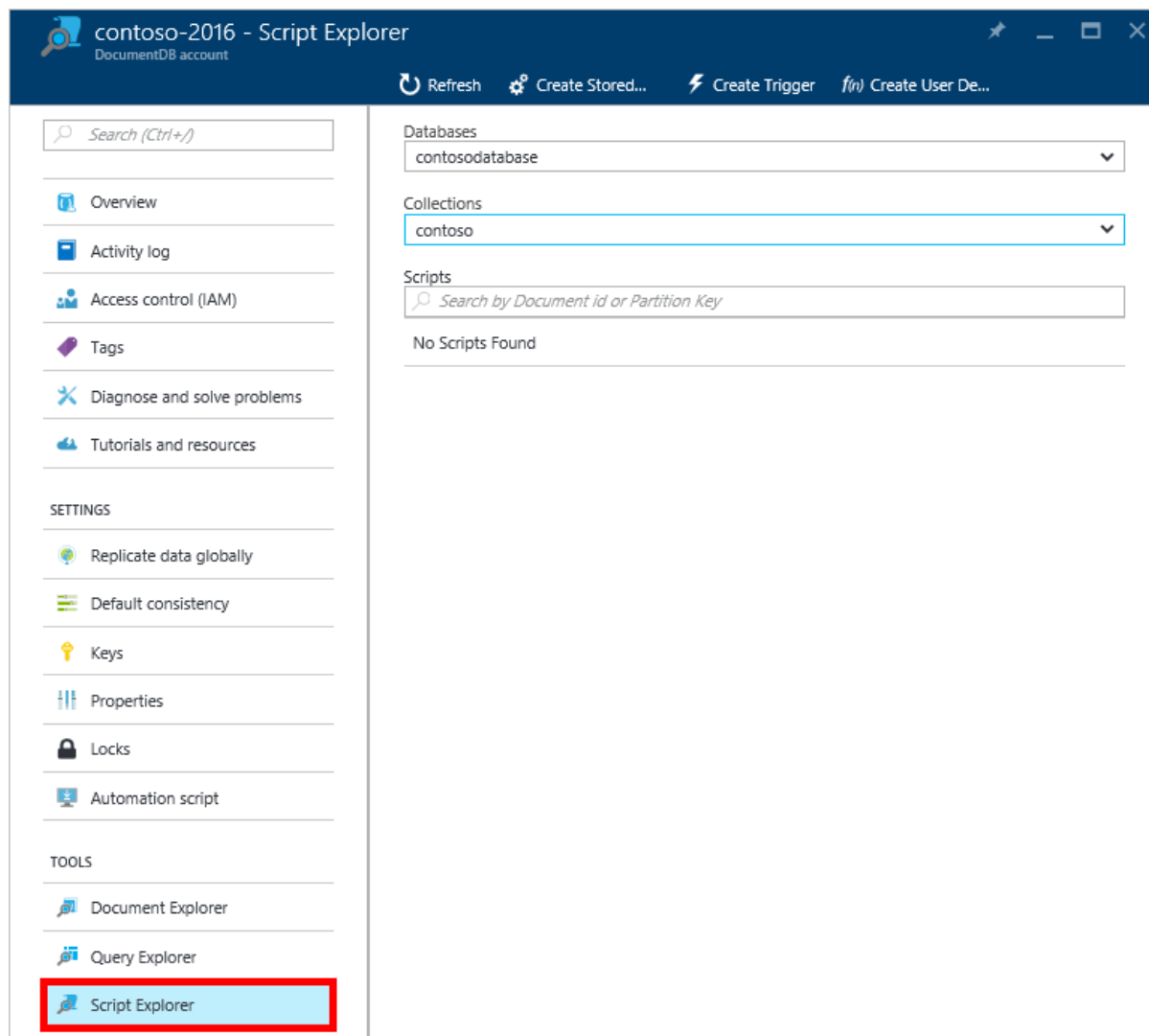
Contributors

Kirill Gavrylyuk • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • mimig • Andrew Hoh • v-aljenk • Stephen Baron • Dene Hager

This article provides an overview of the [Microsoft Azure DocumentDB](#) Script Explorer, which is a JavaScript editor in the Azure portal that enables you to view and execute DocumentDB server-side programming artifacts including stored procedures, triggers, and user-defined functions. Read more about DocumentDB server-side programming in the [Stored procedures, database triggers, and UDFs](#) article.

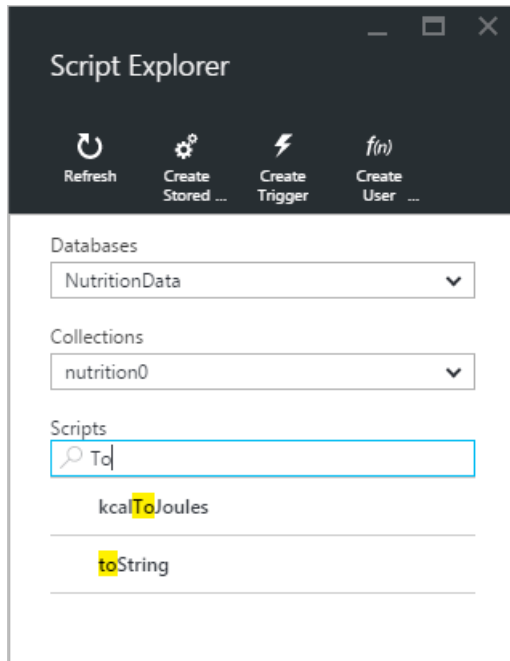
Launch Script Explorer

1. In the Azure portal, in the Jumpbar, click **DocumentDB (NoSQL)**. If **DocumentDB Accounts** is not visible, click **More Services** and then click **DocumentDB (NoSQL)**.
2. In the resources menu, click **Script Explorer**.



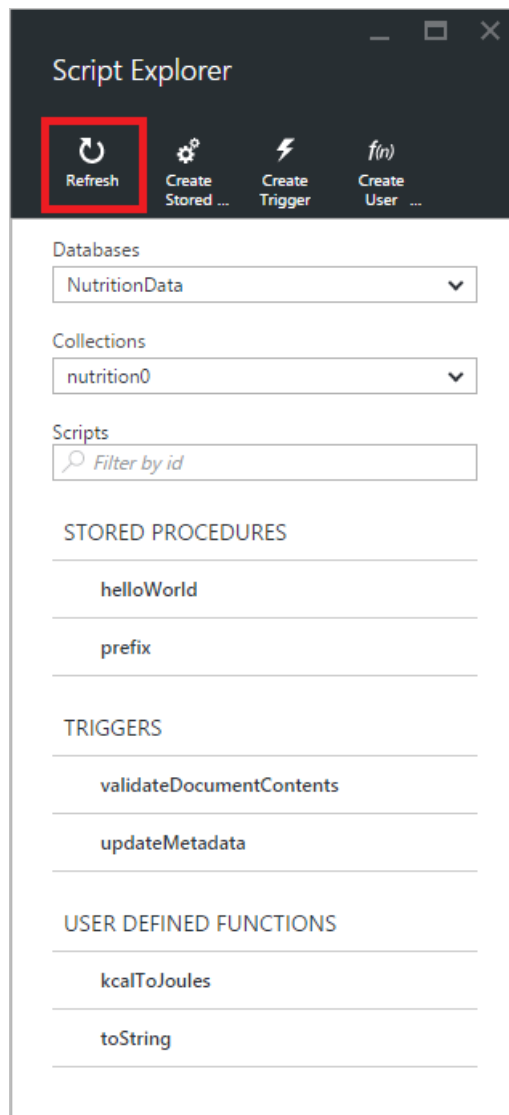
The **Database** and **Collection** drop-down list boxes are pre-populated depending on the context in which you launch Script Explorer. For example, if you launch from a database blade, then the current database is pre-populated. If you launch from a collection blade, then the current collection is pre-populated.

3. Use the **Database** and **Collection** drop-down list boxes to easily change the collection from which scripts are currently being viewed without having to close and re-launch Script Explorer.
4. Script Explorer also supports filtering the currently loaded set of scripts by their id property. Simply type in the filter box and the results in the Script Explorer list are filtered based on your supplied criteria.



[AZURE.IMPORTANT] The Script Explorer filter functionality only filters from the *currently* loaded set of scripts and does not automatically refresh the currently selected collection.

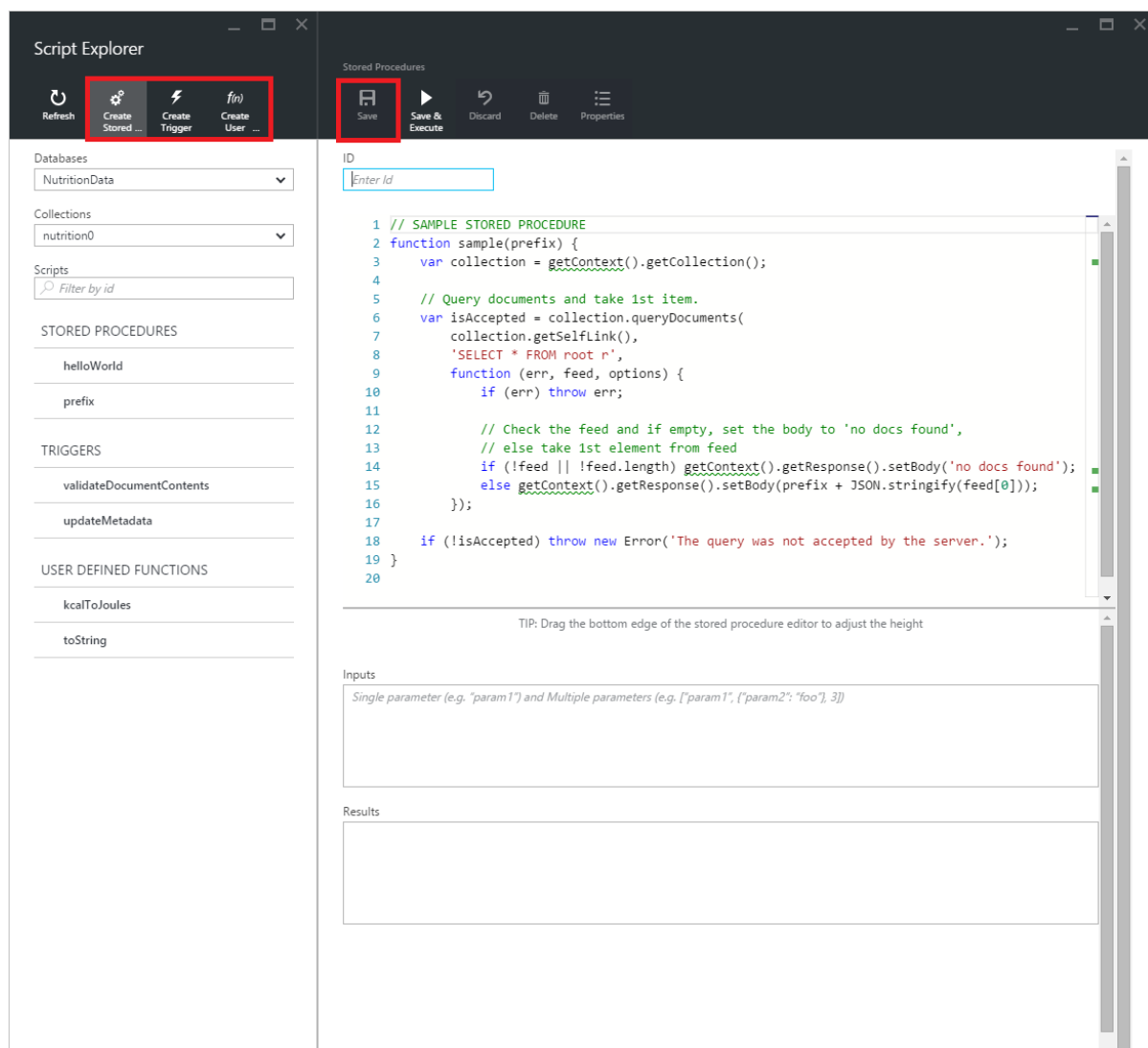
5. To refresh the list of scripts loaded by Script Explorer, simply click the **Refresh** command at the top of the blade.



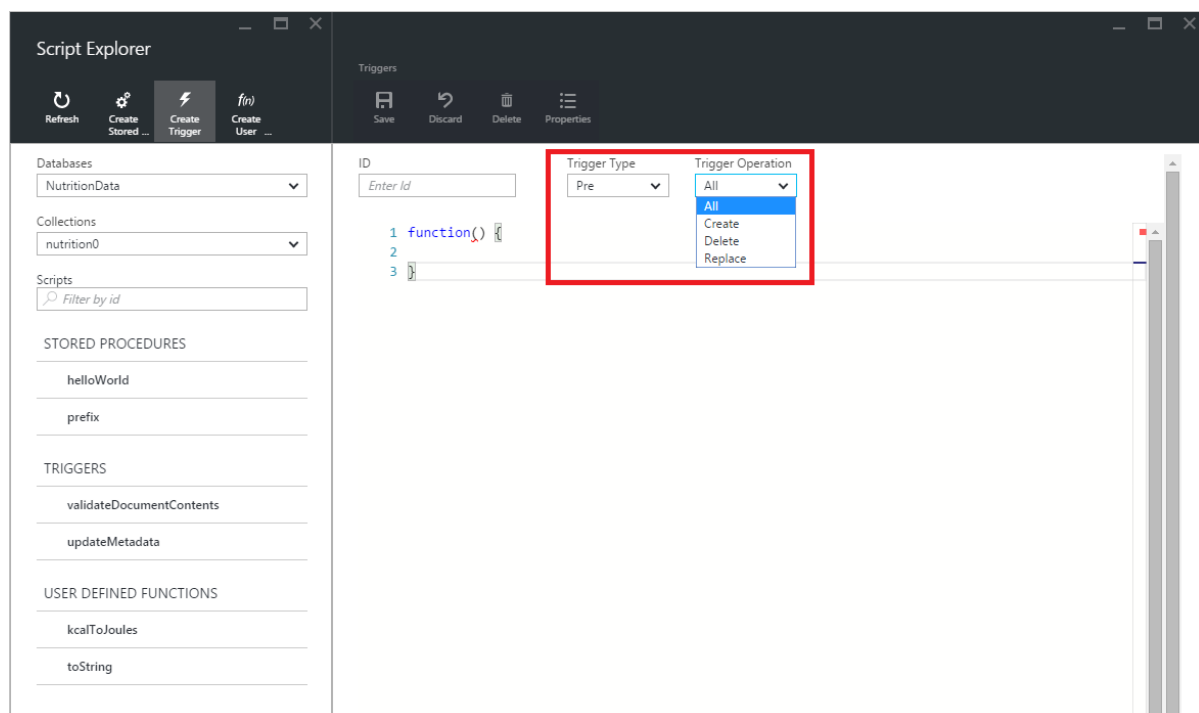
Create, view, and edit stored procedures, triggers, and user-defined functions

Script Explorer allows you to easily perform CRUD operations on DocumentDB server-side programming artifacts.

- To create a script, simply click on the applicable create command within script explorer, provide an id, enter the contents of the script, and click **Save**.



- When creating a trigger, you must also specify the trigger type and trigger operation



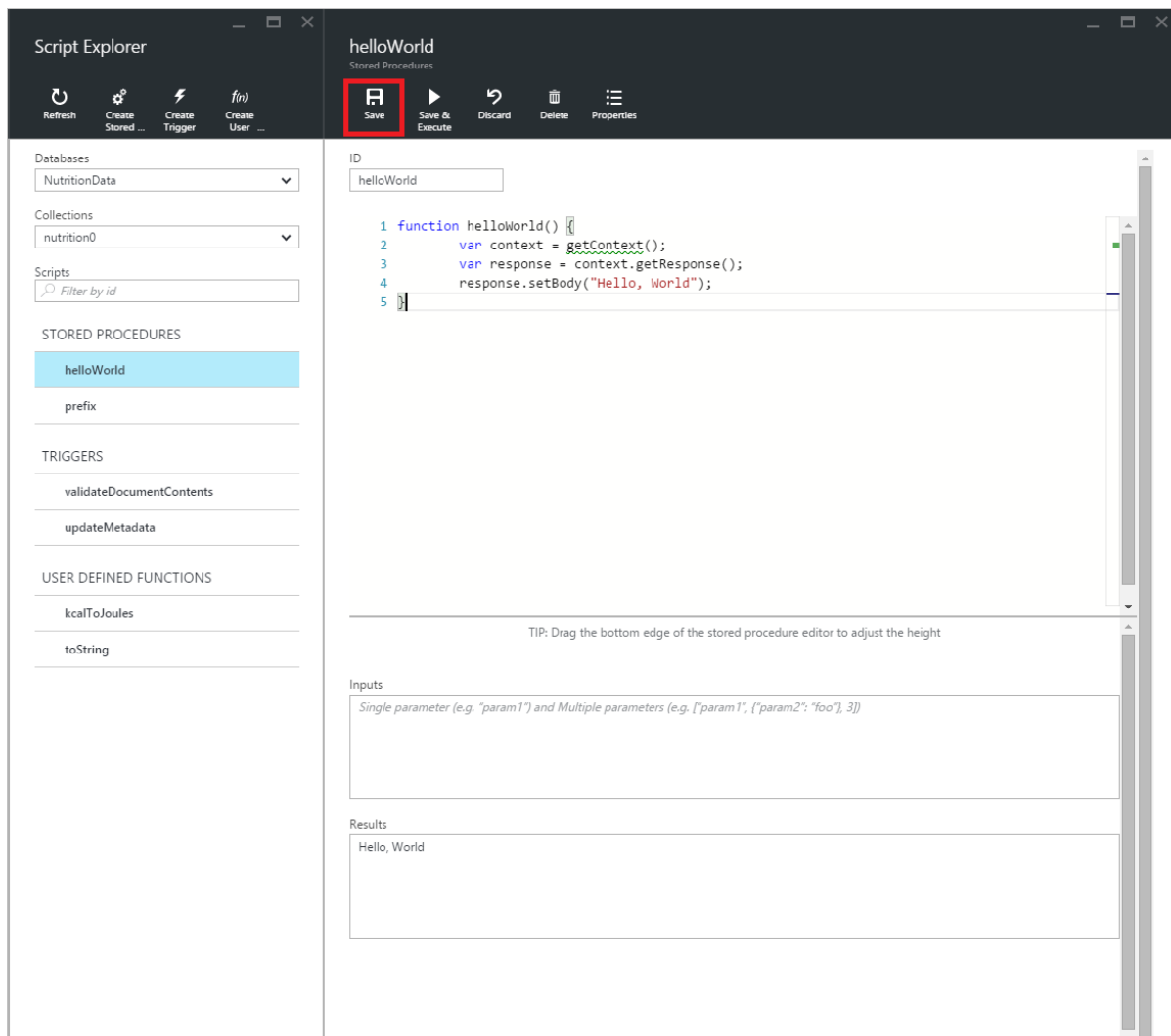
- To view a script, simply click the script in which you're interested.

The screenshot displays the 'Script Explorer' application window. The left sidebar contains a tree view with categories: Databases (NutritionData), Collections (nutrition0), Scripts (Filter by id), STORED PROCEDURES (highlighted with a red box, containing 'helloWorld'), TRIGGERS (validateDocumentContents, updateMetadata), and USER DEFINED FUNCTIONS (kcalToJoules, toString). The main panel is titled 'helloWorld' and shows the JavaScript code for the stored procedure. The code is as follows:

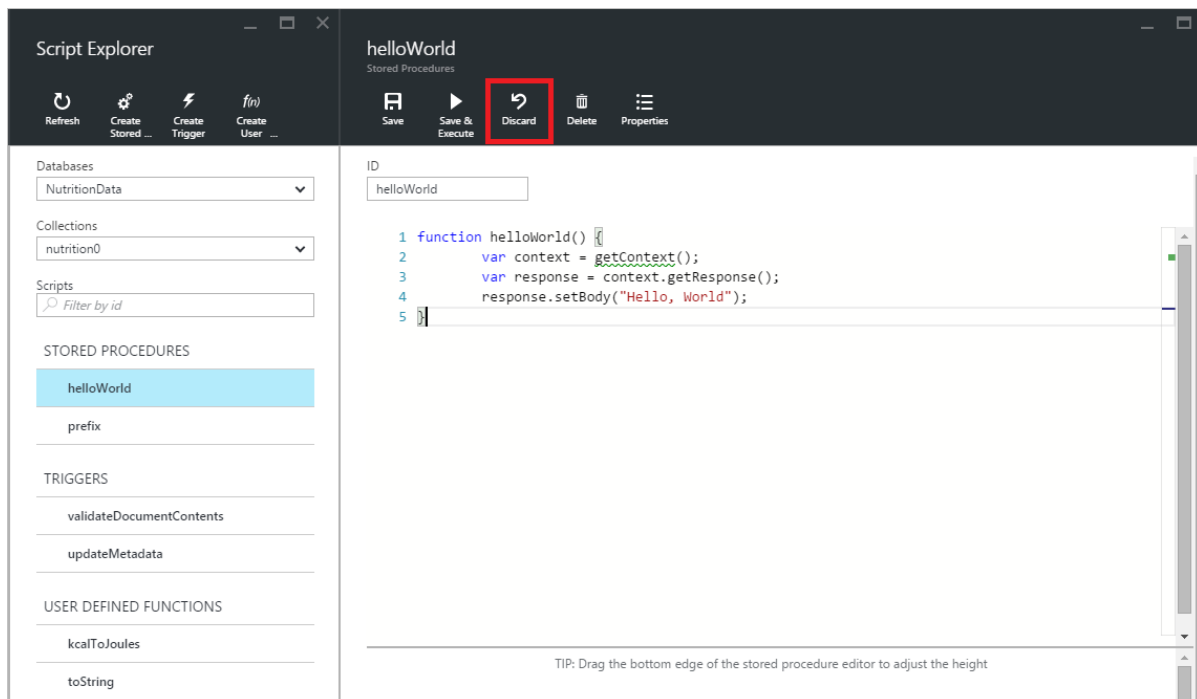
```
1 function helloWorld() {  
2   var context = getContext();  
3   var response = context.getResponse();  
4   response.setBody("Hello, World");  
5 }
```

Below the code editor, there is a 'TIP: Drag the bottom edge of the stored procedure editor to adjust the height' message. At the bottom, there are sections for 'Inputs' (with a placeholder text: 'Single parameter (e.g. "param1") and Multiple parameters (e.g. ["param1", {"param2": "foo"}, 3])') and 'Results' (displaying 'Hello, World').

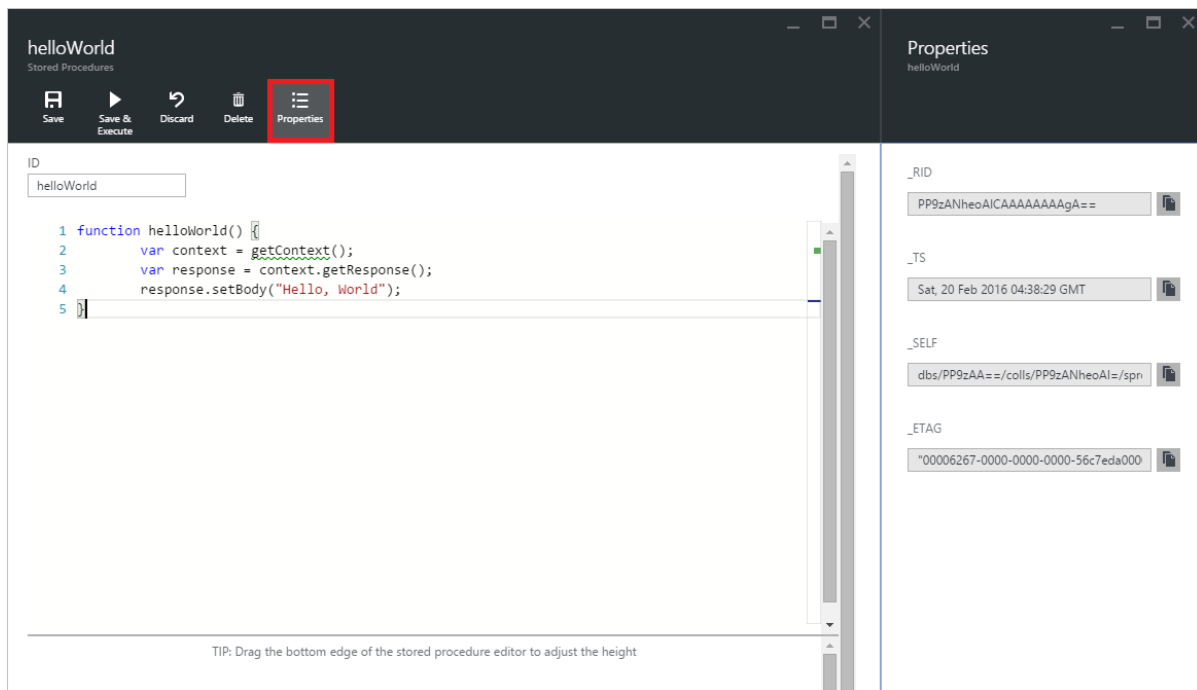
- To edit a script, simply make the desired changes in the JavaScript editor and click **Save**.



- To discard any pending changes to a script, simply click the **Discard** command.



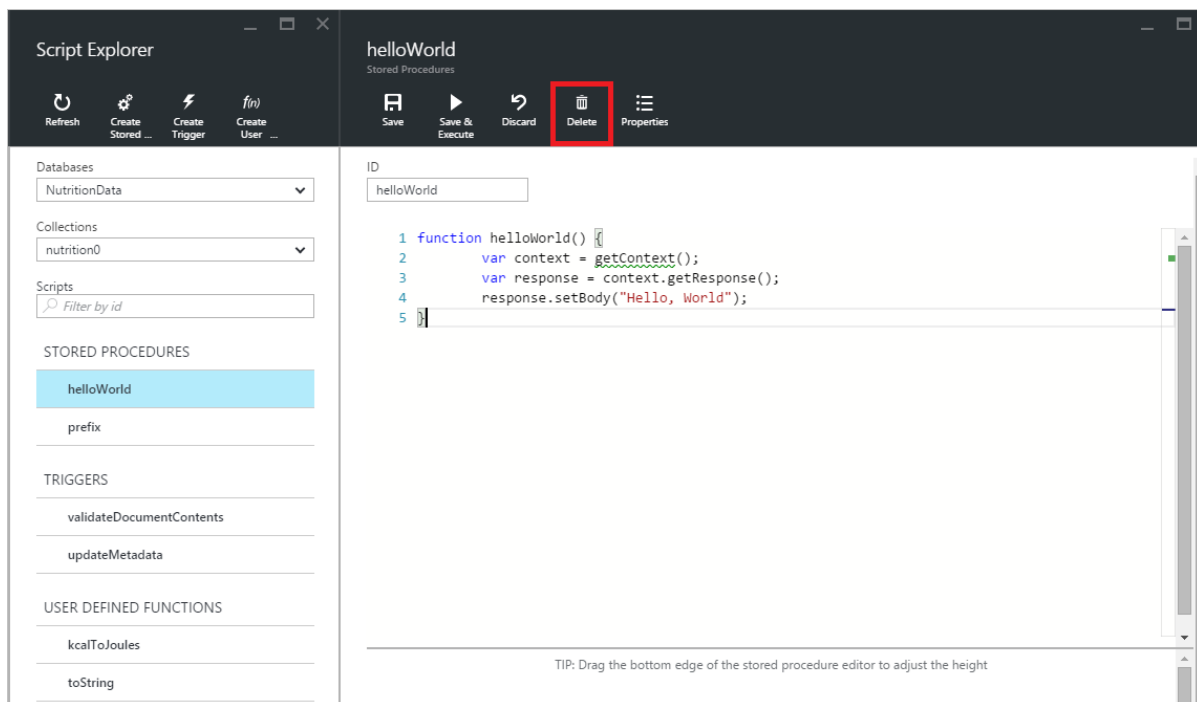
- Script Explorer also allows you to easily view the system properties of the currently loaded script by clicking the **Properties** command.



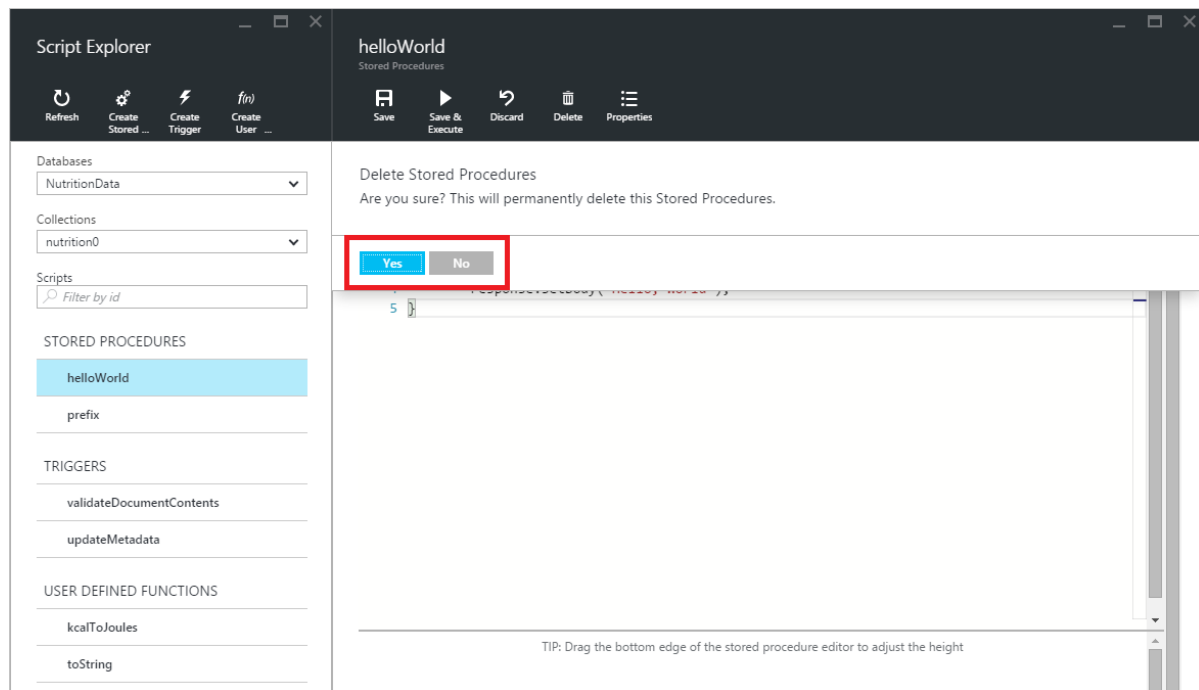
NOTE

The timestamp (_ts) property is internally represented as epoch time, but Script Explorer displays the value in a human readable GMT format.

- To delete a script, select it in Script Explorer and click the **Delete** command.



- Confirm the delete action by clicking **Yes** or cancel the delete action by clicking **No**.



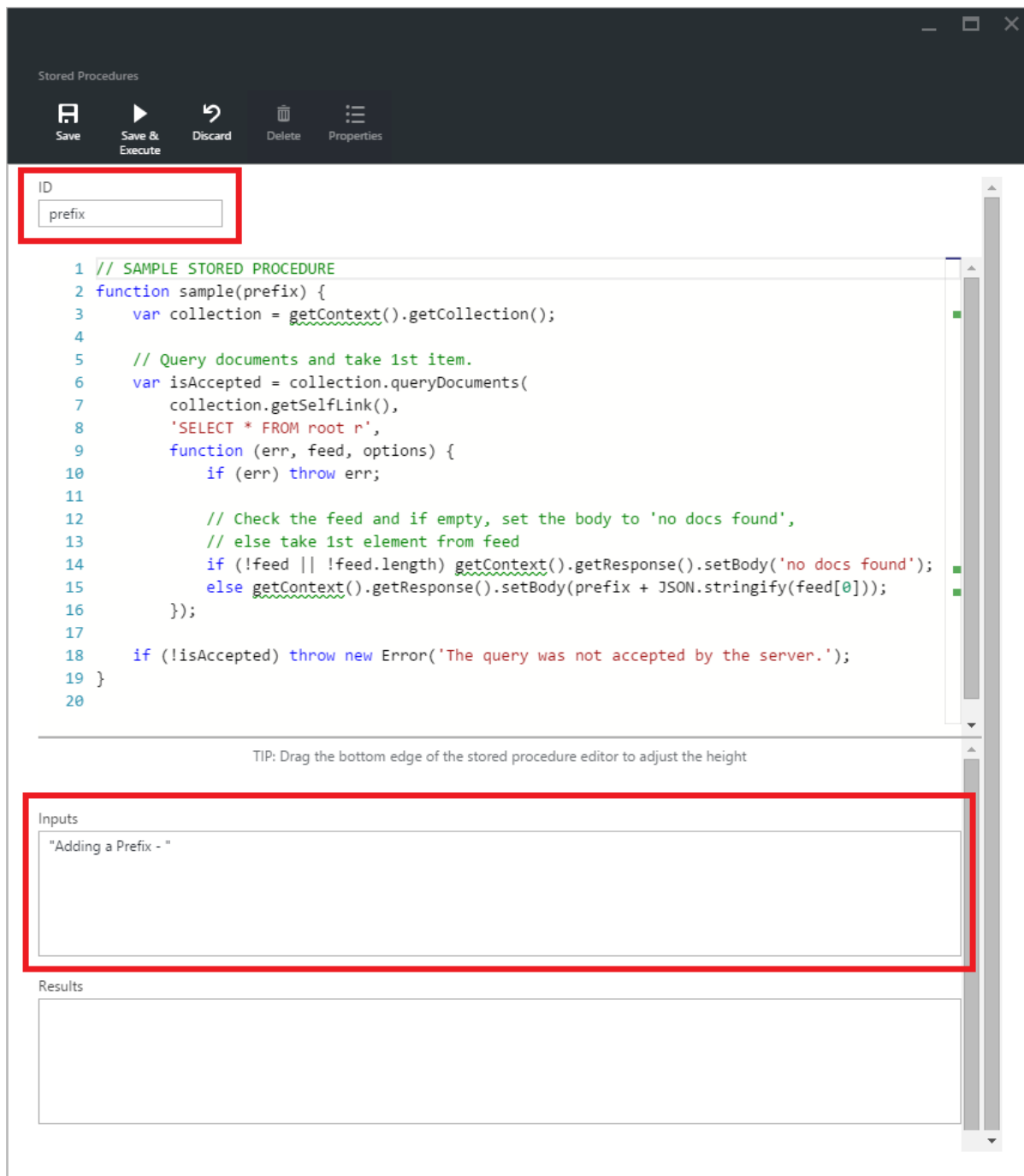
Execute a stored procedure

WARNING

Executing stored procedures in Script Explorer is not yet supported for server side partitioned collections. For more information, visit [Partitioning and Scaling in DocumentDB](#).

Script Explorer allows you to execute server-side stored procedures from the Azure portal.

- When opening a new create stored procedure blade, a default script (*prefix*) will already be provided. In order to run the *prefix* script or your own script, add an *id* and *inputs*. For stored procedures that accept multiple parameters, all inputs must be within an array (e.g. `["foo", "bar"]`).



- To execute a stored procedure, simply click on the **Save & Execute** command within script editor pane.

NOTE

The **Save & Execute** command will save your stored procedure before executing, which means it will overwrite the previously saved version of the stored procedure.

- Successful stored procedure executions will have a *Successfully saved and executed the stored procedure* status and the returned results will be populated in the *Results* pane.

The screenshot shows the Script Explorer application with a dark theme. On the left sidebar, the 'prefix' stored procedure is selected under the 'nutrition0' collection. The main editor displays the code for the 'prefix' stored procedure. The 'Save & Execute' button in the top toolbar is highlighted with a red box. A blue notification bar at the top of the editor area states 'Successfully saved and executed the stored procedure'. Below the code editor, the 'Inputs' section shows the input string 'Adding a Prefix - '. The 'Results' section at the bottom is highlighted with a red box and displays the output: 'Adding a Prefix - [{"foodGroup": "Baby Foods"}]'. A tip below the code editor suggests dragging the bottom edge to adjust the height.

```
1 // SAMPLE STORED PROCEDURE
2 function sample(prefix) {
3   var collection = getContext().getCollection();
4
5   // Query documents and take 1st item.
6   var isAccepted = collection.queryDocuments(
7     collection.getSelfLink(),
8     'SELECT r.foodGroup FROM root r',
9     function (err, feed, options) {
10      if (err) throw err;
11
12      // Check the feed and if empty, set the body to 'no docs found',
13      // else take 1st element from feed
14      if (!feed || !feed.length) getContext().getResponse().setBody('no docs found');
15      else getContext().getResponse().setBody(prefix + JSON.stringify(feed[0]));
16    });
17
18   if (!isAccepted) throw new Error('The query was not accepted by the server.');
```

TIP: Drag the bottom edge of the stored procedure editor to adjust the height

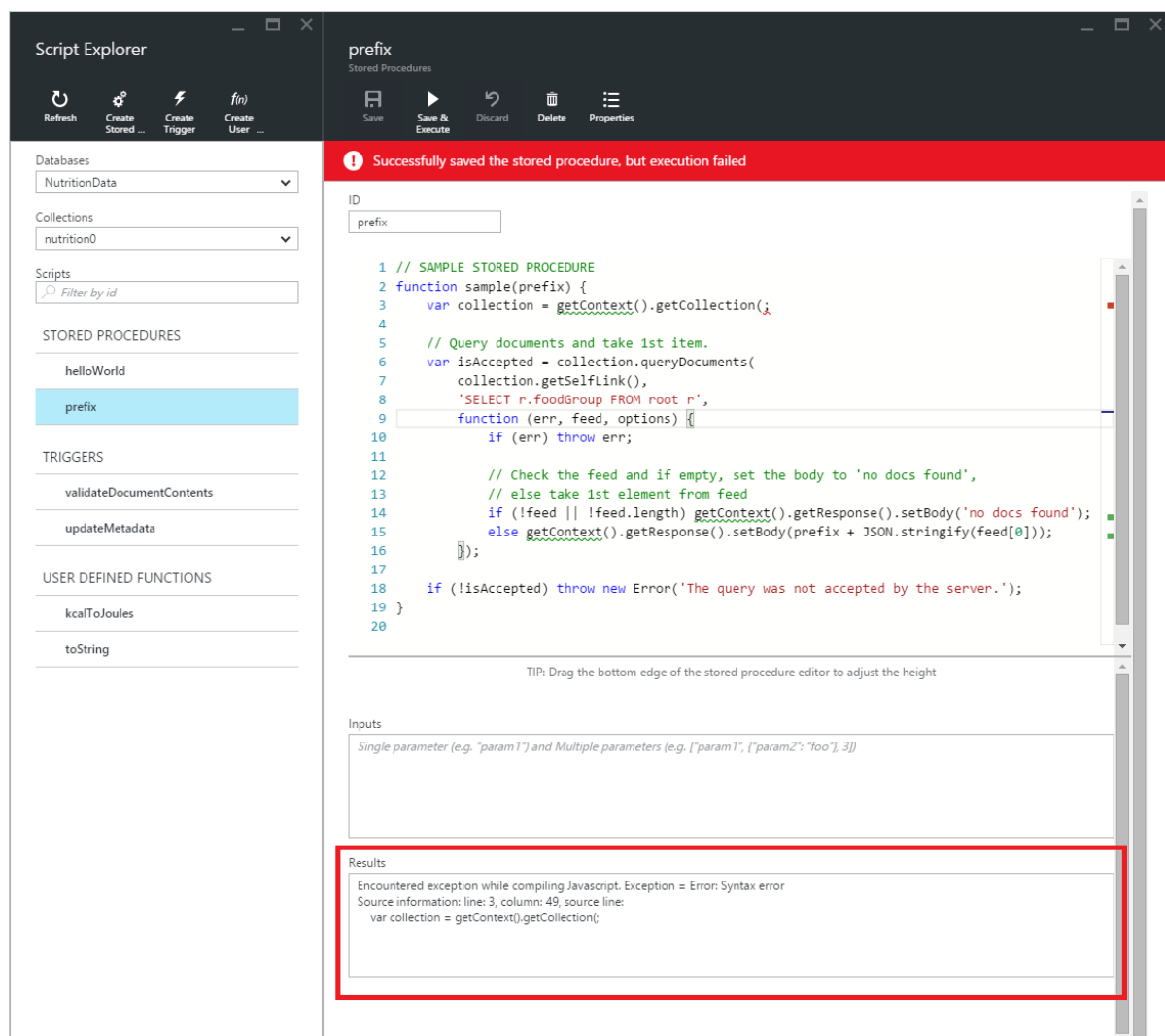
Inputs

"Adding a Prefix - "

Results

Adding a Prefix - [{"foodGroup": "Baby Foods"}]

- If the execution encounters an error, the error will be populated in the *Results* pane.



Work with scripts outside the portal

The Script Explorer in the Azure portal is just one way to work with stored procedures, triggers, and user-defined functions in DocumentDB. You can also work with scripts using the REST API and the [client SDKs](#). The REST API documentation includes samples for working with [stored procedures using REST](#), [user defined functions using REST](#), and [triggers using REST](#). Samples are also available showing how to [work with scripts using C#](#) and [work with scripts using Node.js](#).

Next steps

Learn more about DocumentDB server-side programming in the [Stored procedures, database triggers, and UDFs](#) article.

The [Learning path](#) is also a useful resource to guide you as you learn more about DocumentDB.

Azure DocumentDB portal troubleshooting tips

11/15/2016 • 2 min to read • [Edit on GitHub](#)

Contributors

mimig • [Kim Whitlatch](#) (Beyondsoft Corporation) • [Tyson Nevil](#)

This article describes how to resolve DocumentDB issues in the Azure portal.

Resources are missing

Symptom: Databases or collections are missing from your portal blades.

Solution: Lower application usage to operate under the maximum throughput quota for the collection.

Explanation: The portal is an application like any other, making calls to your DocumentDB database and collection. If your requests are currently being throttled due to calls being made from a separate application, the portal may also be throttled, causing resources not to appear in the portal. To resolve the issue, address the cause of the high throughput usage, and then refresh the portal blade. Information on how to measure and lower throughput usage can be found in the [Throughput](#) section of the [Performance tips](#) article.

Pages or blades won't load

Symptom: Pages and blades in the portal do not display.

Solution: Lower application usage to operate under the maximum throughput quota for the collection.

Explanation: The portal is an application like any other, making calls to your DocumentDB database and collection. If your requests are currently being throttled due to calls being made from a separate application, the portal may also be throttled, causing resources not to appear in the portal. To resolve the issue, address the cause of the high throughput usage, and then refresh the portal blade. Information on how to measure and lower throughput usage can be found in the [Throughput](#) section of the [Performance tips](#) article.

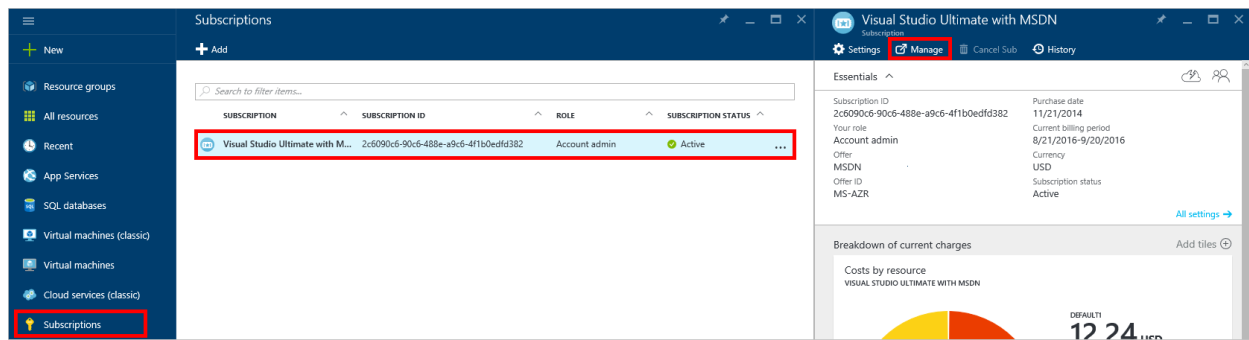
Add Collection button is disabled

Symptom: On the Database blade, the **Add Collection** button is disabled.

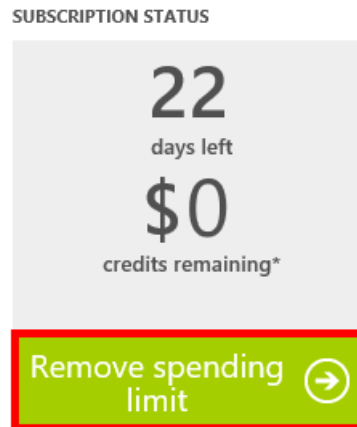
Explanation: If your Azure subscription is associated with benefit credits, such as free credits offered from an MSDN subscription, and you have used all of your credits for the month, you are unable to create any additional collections in DocumentDB.

Solution: Remove the spending limit from your account.

1. In the Azure portal, in the Jumpbar, click **Subscriptions**, click the subscription associated with the DocumentDB database, and then in the **Subscription** blade, click **Manage**.



2. In the new browser window, you'll see that you have no credits remaining. Click the **Remove spending limit** button to remove the spending for only the current billing period or indefinitely. Then complete the wizard to



add or confirm your credit card information.

Query Explorer completes with errors

See [Troubleshoot Query Explorer](#).

No data available in monitoring tiles

See [Troubleshoot monitoring tiles](#).

No documents returned in Document Explorer

See [Troubleshooting Document Explorer](#).

Next steps

If you are still experiencing issues in the portal, please email askdocdb@microsoft.com for assistance, or file a support request in the portal by clicking **Browse, Help + support**, and then clicking **Create support request**.

Deploy DocumentDB and Azure App Service Web Apps using an Azure Resource Manager Template

11/15/2016 • 6 min to read • [Edit on GitHub](#)

Contributors

Han Wong • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • mimig • Stephen Baron • Ryan CrawCour • v-aljenk

This tutorial shows you how to use an Azure Resource Manager template to deploy and integrate [Microsoft Azure DocumentDB](#), an [Azure App Service](#) web app, and a sample web application.

Using Azure Resource Manager templates, you can easily automate the deployment and configuration of your Azure resources. This tutorial shows how to deploy a web application and automatically configure DocumentDB account connection information.

After completing this tutorial, you will be able to answer the following questions:

- How can I use an Azure Resource Manager template to deploy and integrate a DocumentDB account and a web app in Azure App Service?
- How can I use an Azure Resource Manager template to deploy and integrate a DocumentDB account, a web app in App Service Web Apps, and a Webdeploy application?

Prerequisites

TIP

While this tutorial does not assume prior experience with Azure Resource Manager templates or JSON, should you wish to modify the referenced templates or deployment options, then knowledge of each of these areas will be required.

Before following the instructions in this tutorial, ensure that you have the following:

- An Azure subscription. Azure is a subscription-based platform. For more information about obtaining a subscription, see [Purchase Options](#), [Member Offers](#), or [Free Trial](#).

Step 1: Download the template files

Let's start by downloading the template files we will use in this tutorial.

1. Download the [Create a DocumentDB account, Web Apps, and deploy a demo application sample](#) template to a local folder (e.g. C:\DocumentDBTemplates). This template will deploy a DocumentDB account, an App Service web app, and a web application. It will also automatically configure the web application to connect to the DocumentDB account.
2. Download the [Create a DocumentDB account and Web Apps sample](#) template to a local folder (e.g. C:\DocumentDBTemplates). This template will deploy a DocumentDB account, an App Service web app, and will modify the site's application settings to easily surface DocumentDB connection information, but does not include a web application.

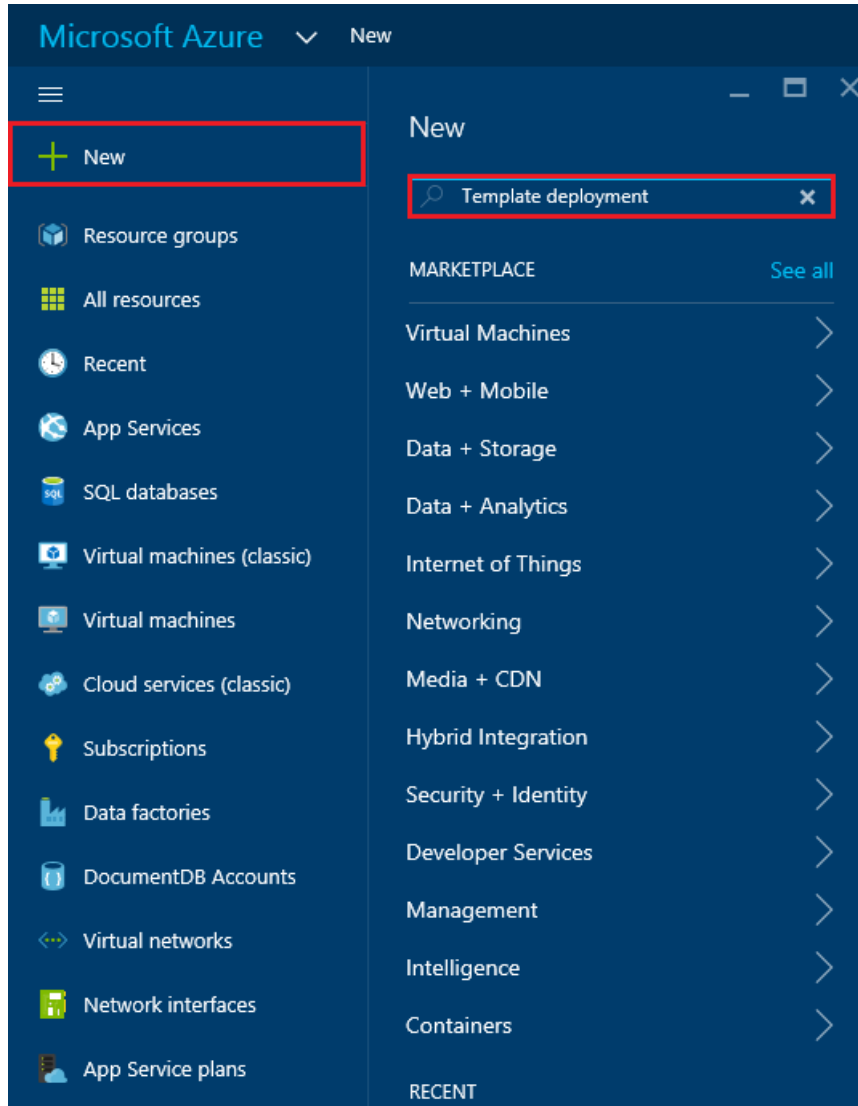
Step 2: Deploy the DocumentDB account, App Service web app and demo application sample

Now let's deploy our first template.

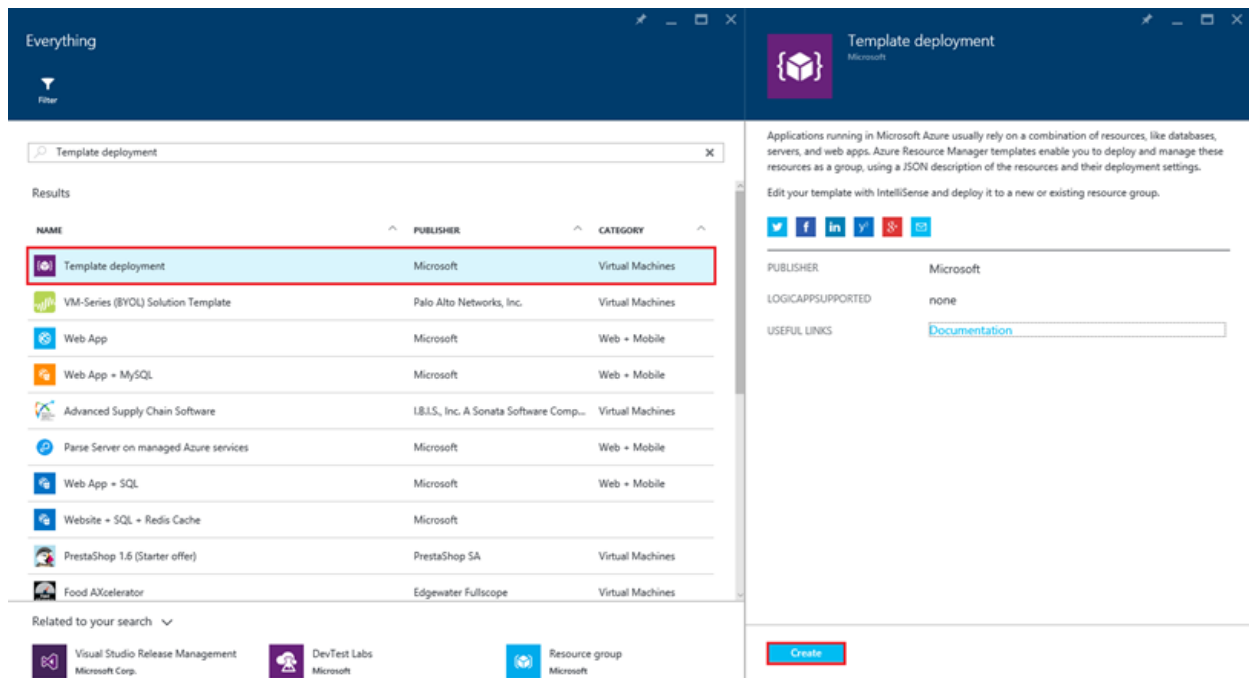
TIP

The template does not validate that the web app name and DocumentDB account name entered below are a) valid and b) available. It is highly recommended that you verify the availability of the names you plan to supply prior to submitting the deployment.

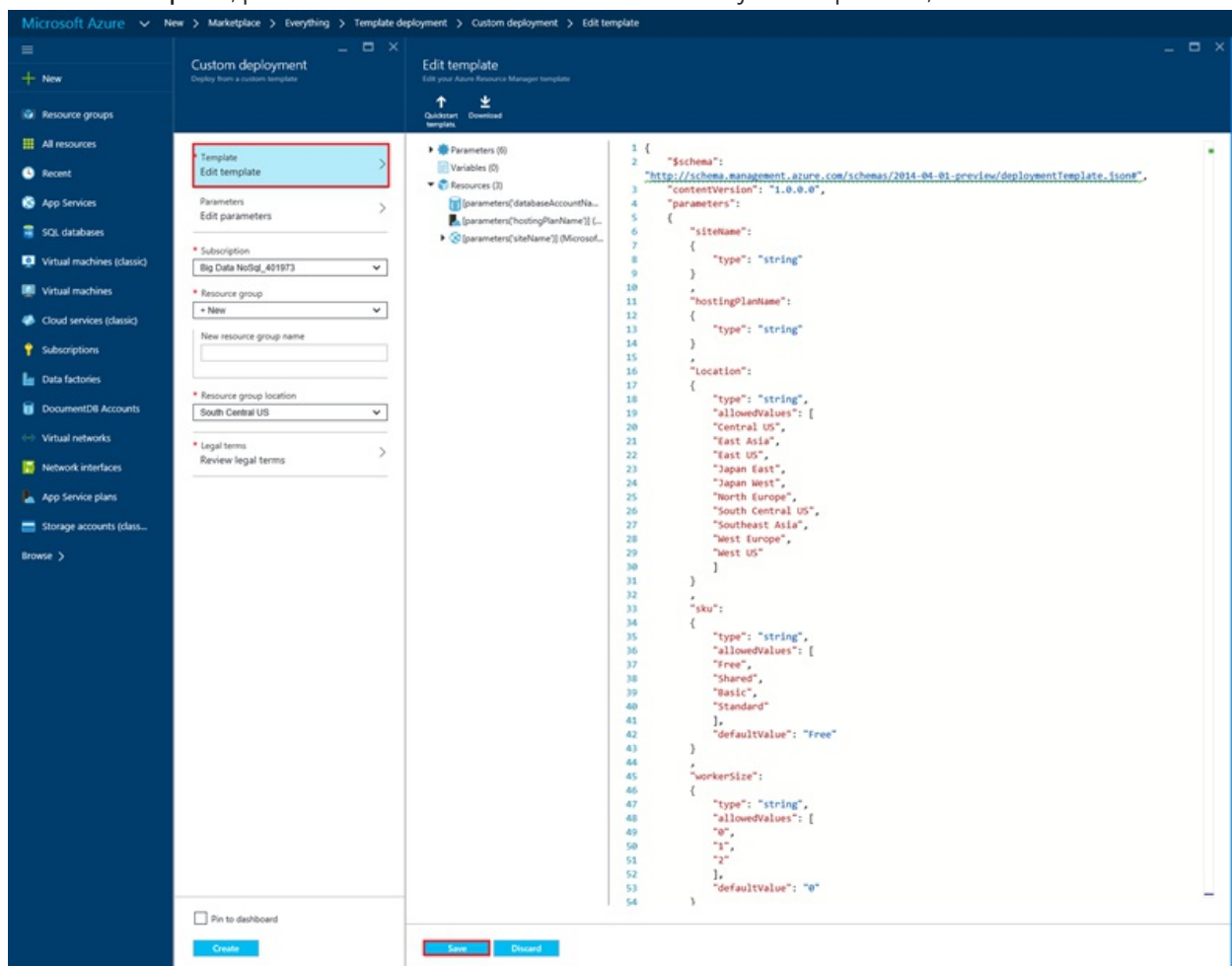
1. Login to the [Azure Portal](#), click New and search for "Template deployment".



2. Select the Template deployment item and click **Create**



- Click **Edit template**, paste the contents of the DocDBWebsiteTodo.json template file, and click **Save**.



- Click **Edit parameters**, provide values for each of the mandatory parameters, and click **OK**. The parameters are as follows:
 - SITENAME**: Specifies the App Service web app name and is used to construct the URL that you will use to access the web app (e.g. if you specify "mydemodocdbwebapp", then the URL by which you will access the web app will be mydemodocdbwebapp.azurewebsites.net).
 - HOSTINGPLANNAME**: Specifies the name of App Service hosting plan to create.
 - LOCATION**: Specifies the Azure location in which to create the DocumentDB and web app resources.
 - DATABASEACCOUNTNAME**: Specifies the name of the DocumentDB account to create.

Microsoft Azure > New > Marketplace > Everything > Template deployment > Custom deployment > Parameters

New

Resource groups

All resources

Recent

App Services

SQL databases

Virtual machines (classic)

Virtual machines

Cloud services (classic)

Subscriptions

Data factories

DocumentDB Accounts

Virtual networks

Network interfaces

App Service plans

Storage accounts (class...

Browse >

Custom deployment

Deploy from a custom template

* Template

Edit template

Parameters

Edit parameters

* Subscription

Big Data NoSql_401973

* Resource group

+ New

New resource group name

* Resource group location

South Central US

* Legal terms

Review legal terms

☐ Pin to dashboard

Create

Parameters

Customize your template parameters

* SITENAME (string)

mydemotodoappsite

* HOSTINGPLANNAME (string)

mydemotodoplan

* LOCATION (string)

Central US

* DATABASEACCOUNTNAME (string)

mydemotodocdb

OK

5. Choose an existing Resource group or provide a name to make a new resource group, and choose a location for

Microsoft Azure

New > Marketplace > Everything > Template de

Custom deployment
Deploy from a custom template

* Template
Edit template

* Parameters
Edit parameters

* Subscription
Big Data NoSql_401973

* Resource group
+ New
New resource group name
MyNewResourceGroup

* Resource group location
South Central US

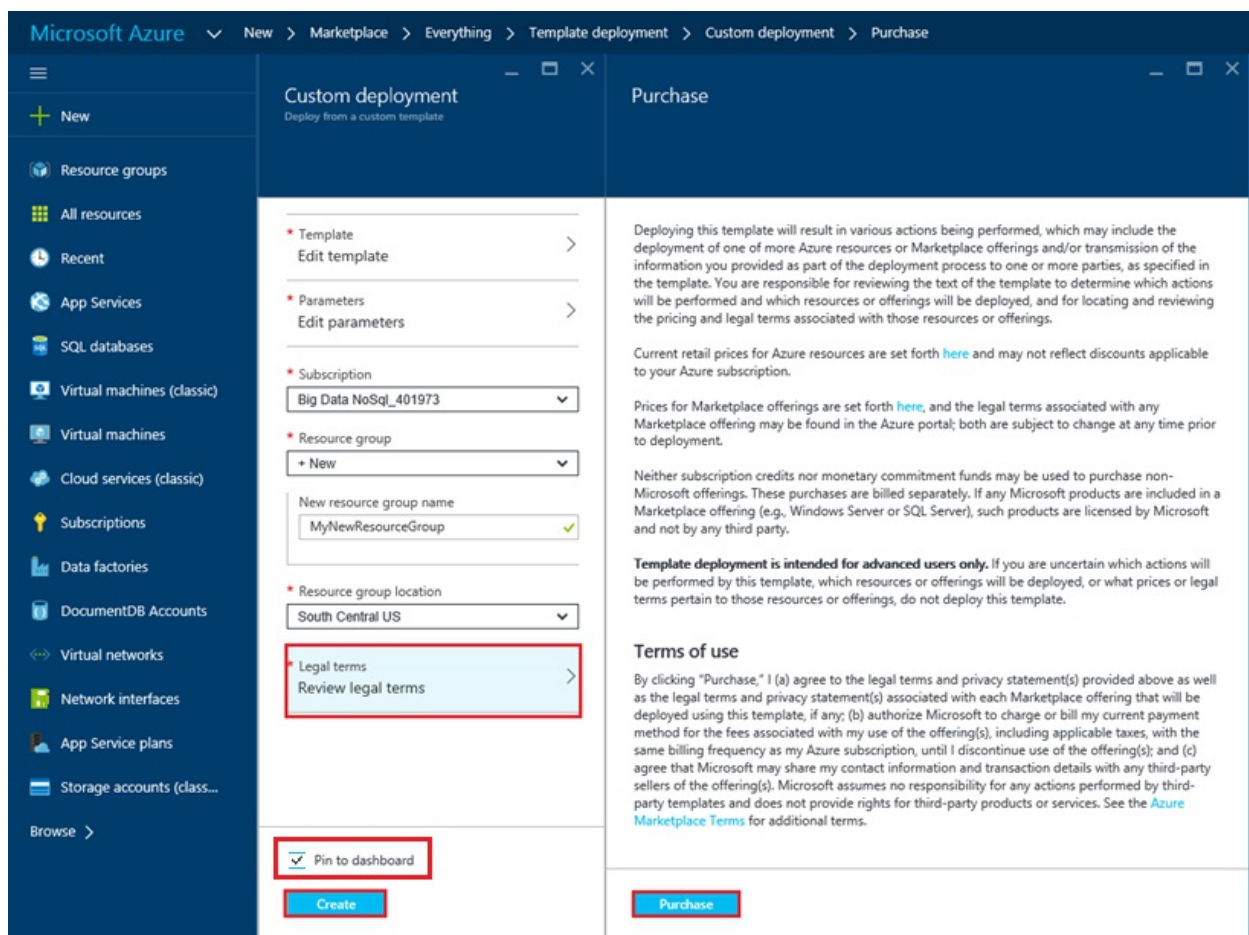
* Legal terms
Review legal terms

☐ Pin to dashboard

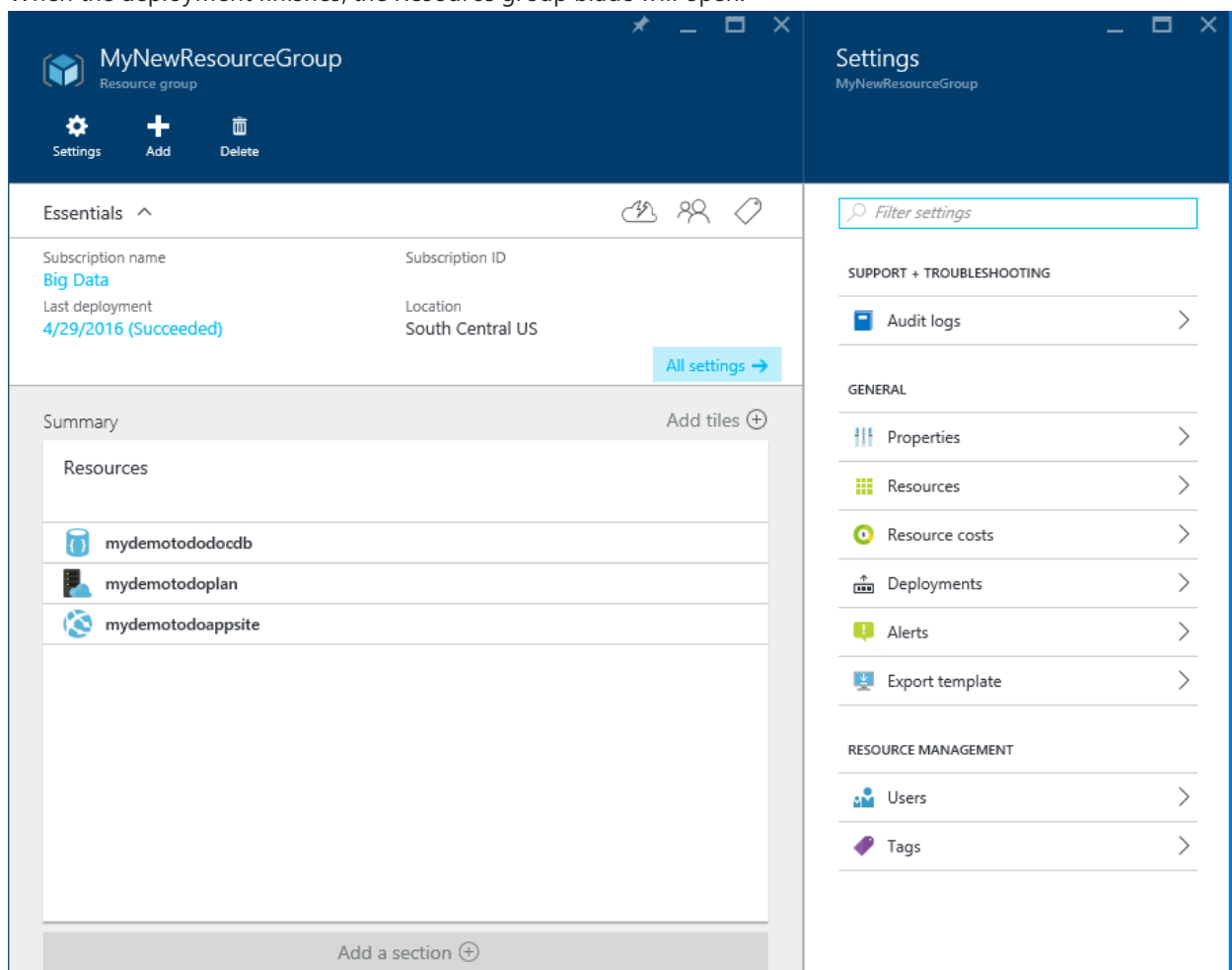
Create

the resource group.

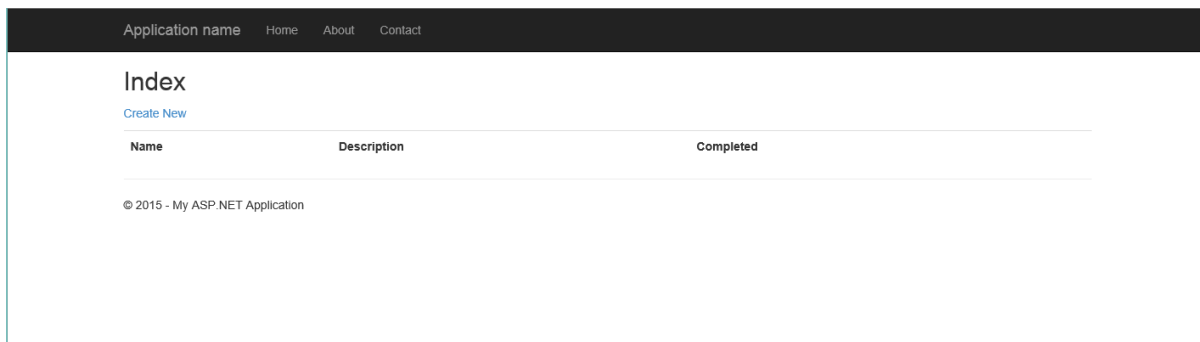
6. Click **Review legal terms**, **Purchase**, and then click **Create** to begin the deployment. Select **Pin to dashboard** so the resulting deployment is easily visible on your Azure portal home page.



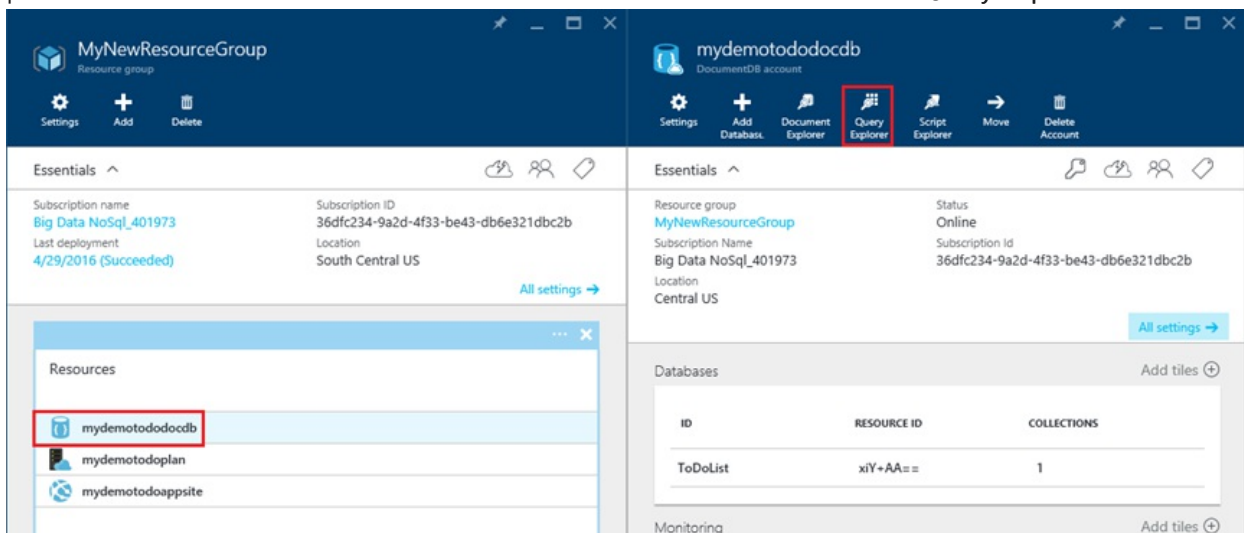
7. When the deployment finishes, the Resource group blade will open.



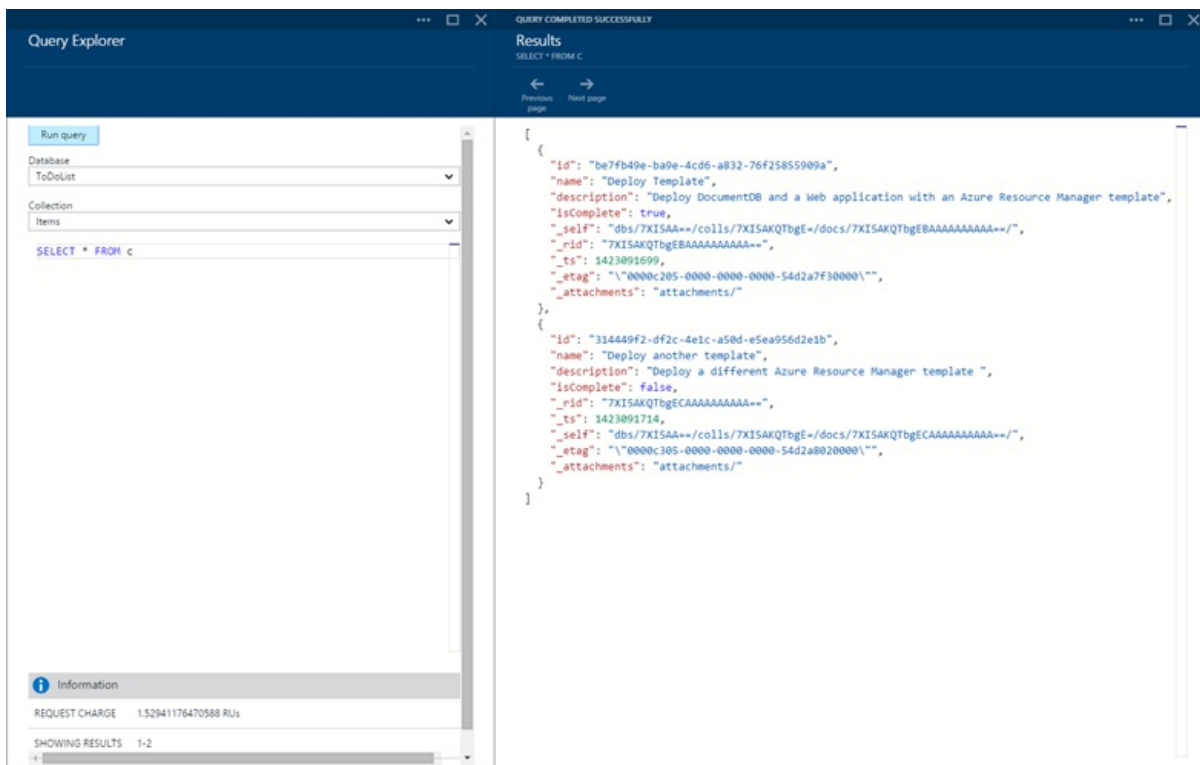
8. To use the application, simply navigate to the web app URL (in the example above, the URL would be <http://mydemodocdbwebapp.azurewebsites.net>). You'll see the following web application:



9. Go ahead and create a couple of tasks in the web app and then return to the Resource group blade in the Azure portal. Click the DocumentDB account resource in the Resources list and then click **Query Explorer**.



10. Run the default query, "SELECT * FROM c" and inspect the results. Notice that the query has retrieved the JSON representation of the todo items you created in step 7 above. Feel free to experiment with queries; for example, try running SELECT * FROM c WHERE c.isComplete = true to return all todo items which have been marked as complete.



11. Feel free to explore the DocumentDB portal experience or modify the sample Todo application. When you're ready, let's deploy another template.

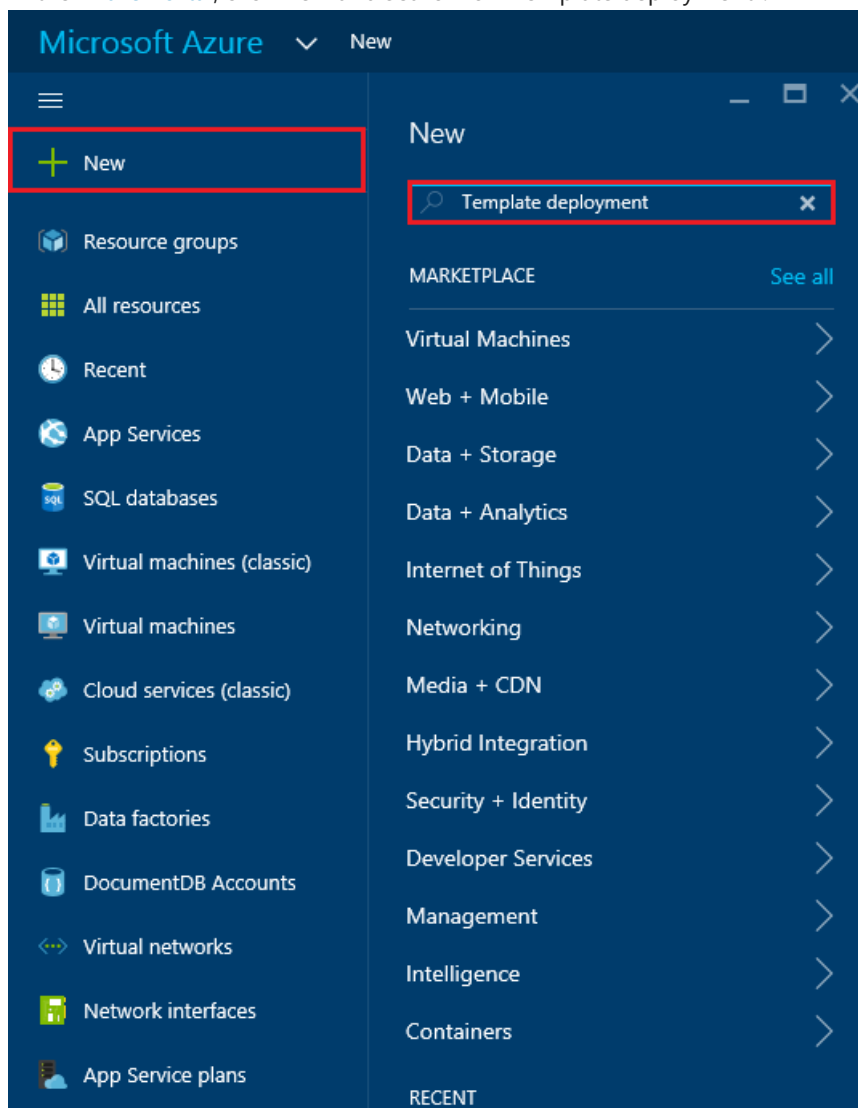
Step 3: Deploy the Document account and web app sample

Now let's deploy our second template. This template is useful to show how you can inject DocumentDB connection information such as account endpoint and master key into a web app as application settings or as a custom connection string. For example, perhaps you have your own web application that you would like to deploy with a DocumentDB account and have the connection information automatically populated during deployment.

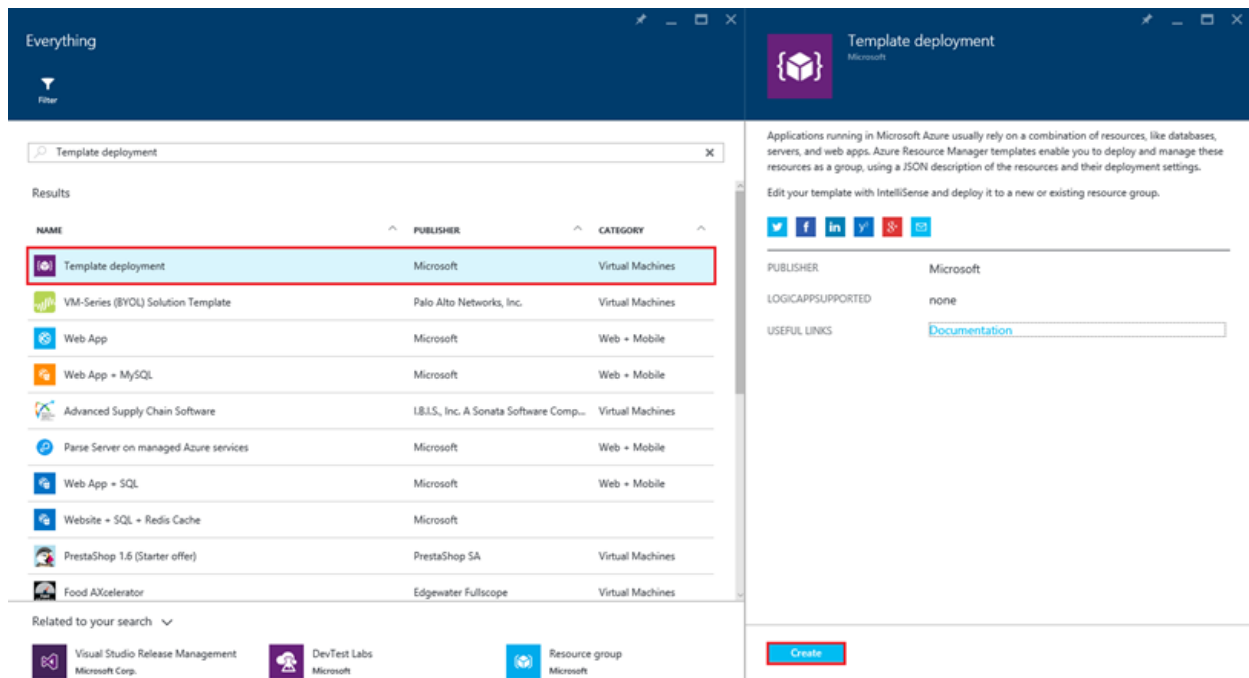
TIP

The template does not validate that the web app name and DocumentDB account name entered below are a) valid and b) available. It is highly recommended that you verify the availability of the names you plan to supply prior to submitting the deployment.

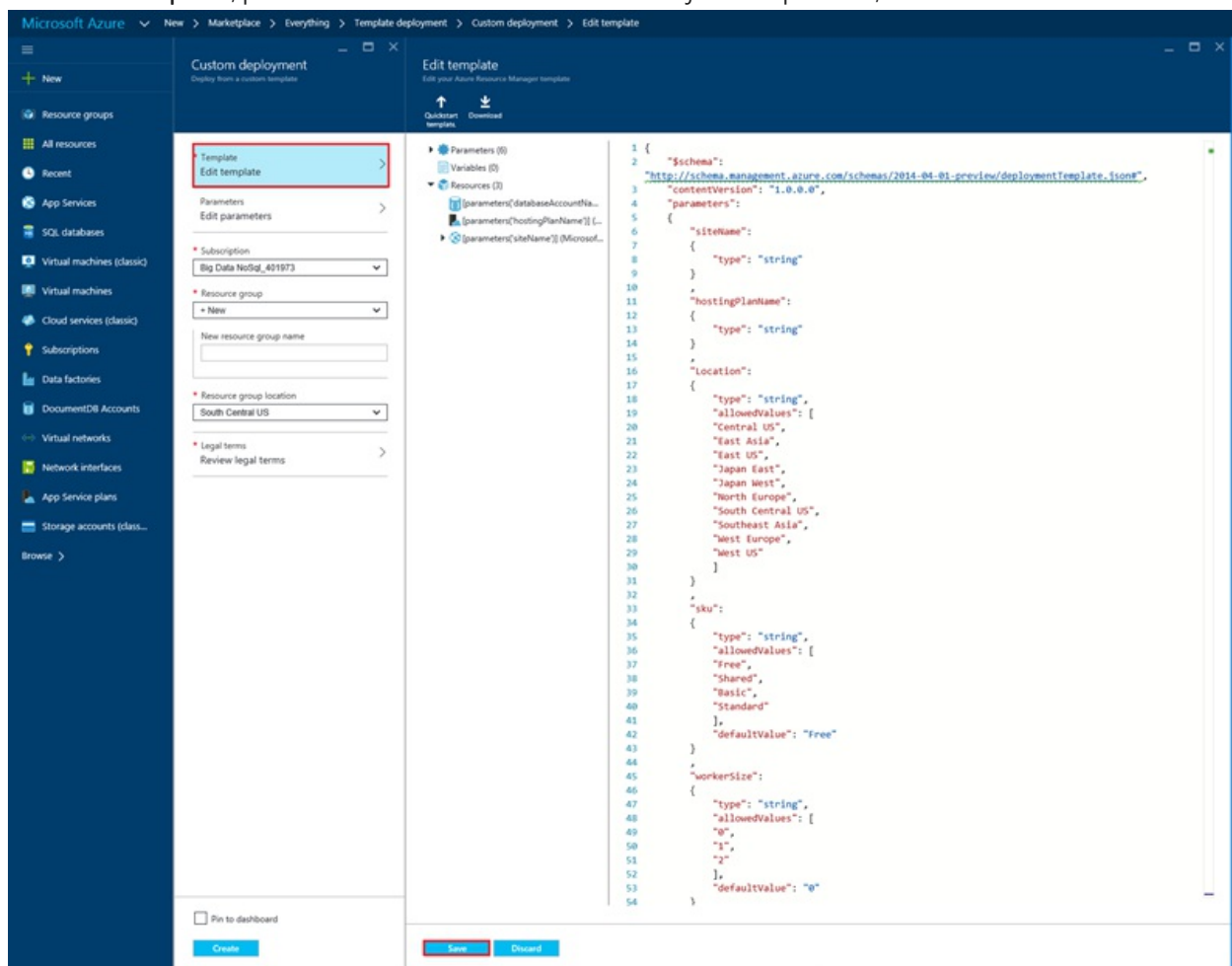
1. In the [Azure Portal](#), click New and search for "Template deployment".



2. Select the Template deployment item and click **Create**



3. Click **Edit template**, paste the contents of the DocDBWebSite.json template file, and click **Save**.



4. Click **Edit parameters**, provide values for each of the mandatory parameters, and click **OK**. The parameters are as follows:
 - a. **SITENAME**: Specifies the App Service web app name and is used to construct the URL that you will use to access the web app (e.g. if you specify "mydemodocdbwebapp", then the URL by which you will access the web app will be mydemodocdbwebapp.azurewebsites.net).
 - b. **HOSTINGPLANNAME**: Specifies the name of App Service hosting plan to create.
 - c. **LOCATION**: Specifies the Azure location in which to create the DocumentDB and web app resources.
 - d. **DATABASEACCOUNTNAME**: Specifies the name of the DocumentDB account to create.

Microsoft Azure > New > Marketplace > Everything > Template deployment > Custom deployment > Parameters

New

Resource groups

All resources

Recent

App Services

SQL databases

Virtual machines (classic)

Virtual machines

Cloud services (classic)

Subscriptions

Data factories

DocumentDB Accounts

Virtual networks

Network interfaces

App Service plans

Storage accounts (class...

Browse >

Custom deployment

Deploy from a custom template

* Template

Edit template

Parameters

Edit parameters

* Subscription

Big Data NoSql_401973

* Resource group

+ New

New resource group name

* Resource group location

South Central US

* Legal terms

Review legal terms

☐ Pin to dashboard

Create

Parameters

Customize your template parameters

* SITENAME (string)

mydemotodoappsite

* HOSTINGPLANNAME (string)

mydemotodoplan

* LOCATION (string)

Central US

* DATABASEACCOUNTNAME (string)

mydemotodocdb

OK

5. Choose an existing Resource group or provide a name to make a new resource group, and choose a location for

Microsoft Azure

New > Marketplace > Everything > Template de

Custom deployment
Deploy from a custom template

* Template
Edit template

* Parameters
Edit parameters

* Subscription
Big Data NoSql_401973

* Resource group
+ New
New resource group name
MyNewResourceGroup

* Resource group location
South Central US

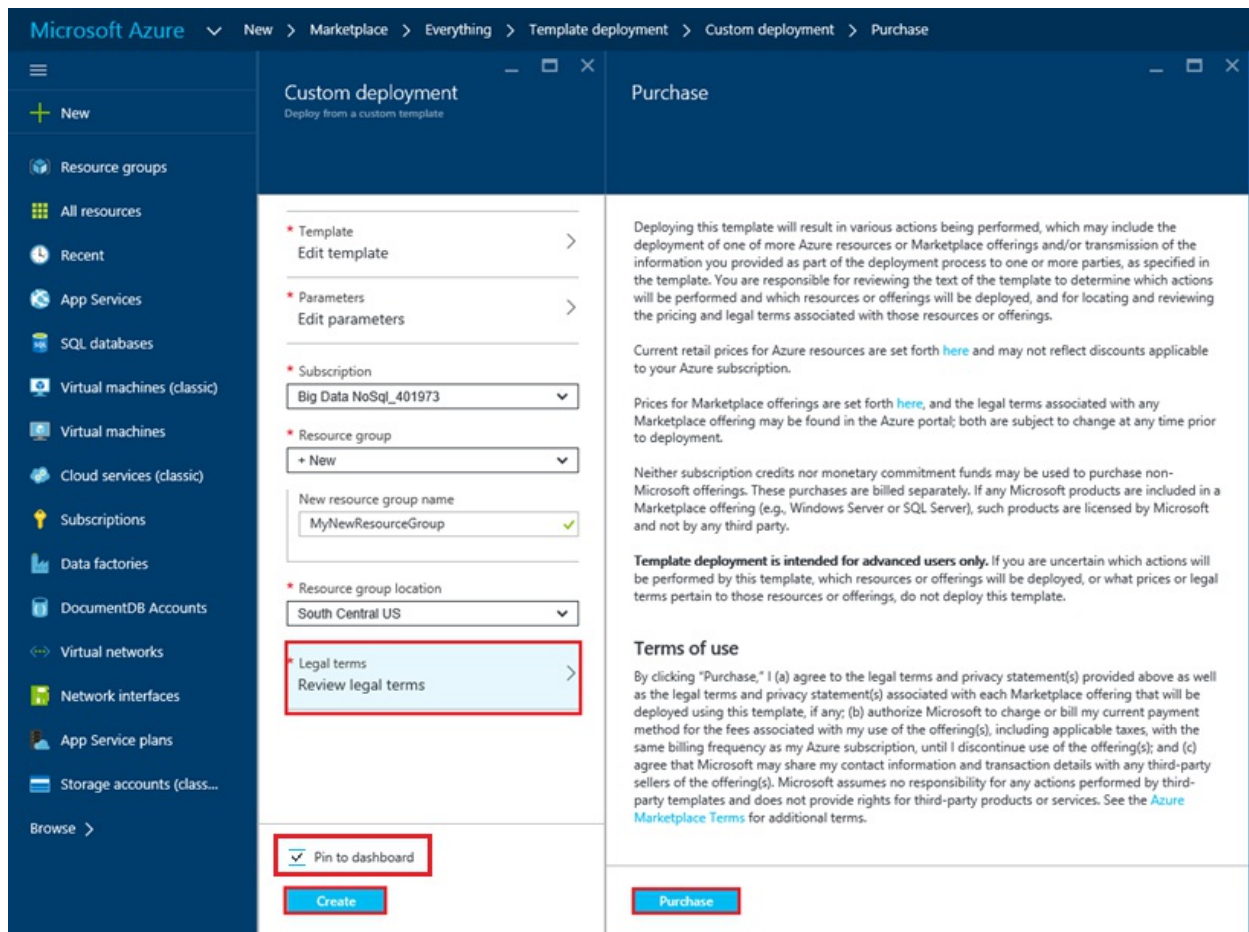
* Legal terms
Review legal terms

☐ Pin to dashboard

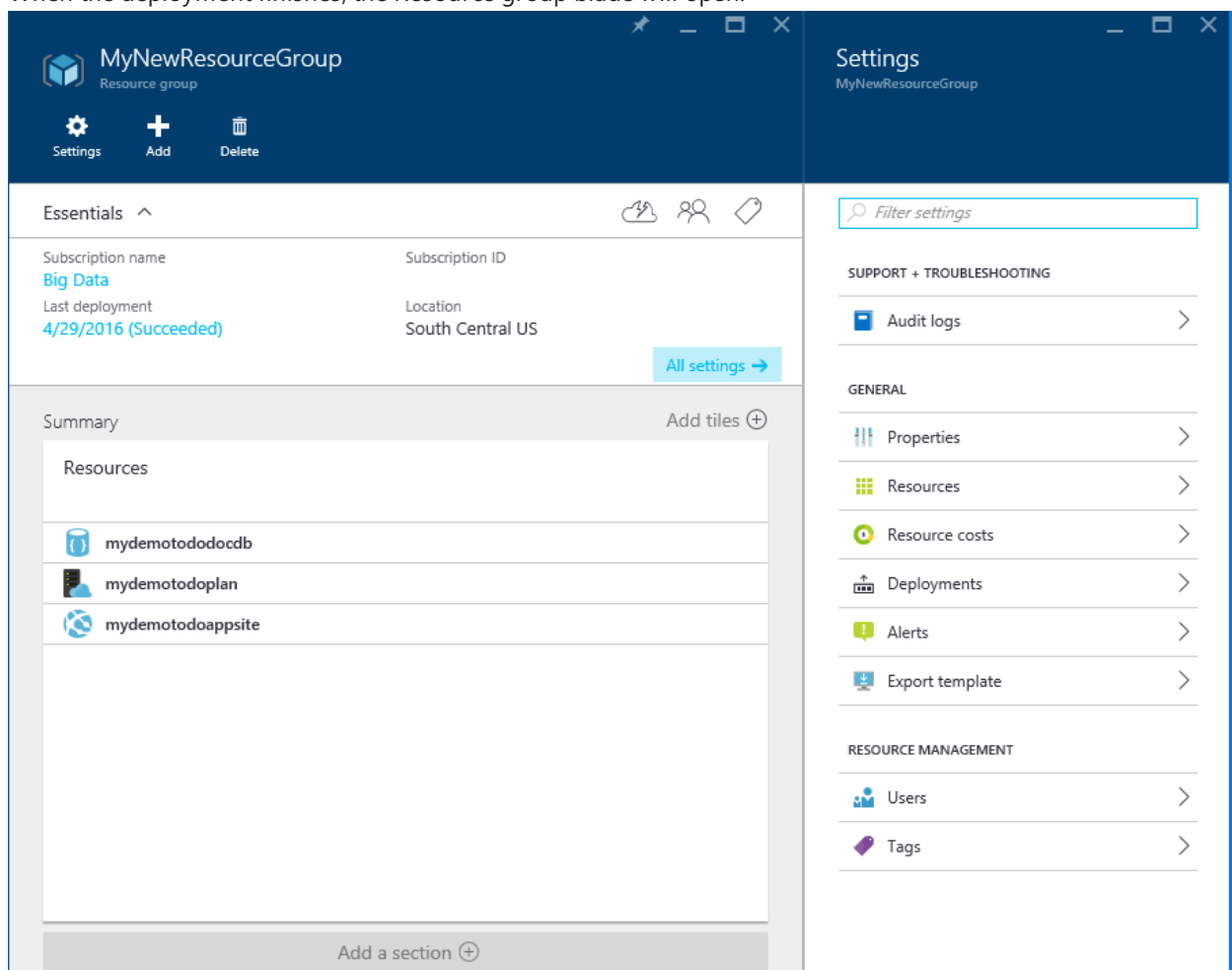
Create

the resource group.

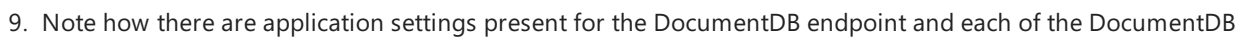
6. Click **Review legal terms**, **Purchase**, and then click **Create** to begin the deployment. Select **Pin to dashboard** so the resulting deployment is easily visible on your Azure portal home page.



7. When the deployment finishes, the Resource group blade will open.



8. Click the Web App resource in the Resources list and then click **Application settings**



Application settings

mydocdbwebapp

Save

Discard

General settings

The 64-bit and Always On options can be enabled in Basic plans and higher. AutoSwap can only be enabled on Standard plans.

.NET Framework version ⓘ

v4.6

PHP version ⓘ

5.4

Java version ⓘ

Off

Python version ⓘ

Off

Platform ⓘ

32-bit64-bit

Web sockets ⓘ

OffOn

Always On ⓘ

OffOn

Managed Pipeline Version

IntegratedClassic

Auto swap destinations cannot be configured from production slot

Auto Swap

OffOn

Auto Swap Slot

Debugging

Remote debugging

OffOn

Remote Visual Studio version

201220132015

App settings

endpoint

https://mydocdbwebaccct.d...

Slot setting

...

authKey

I4cUDBeTw2rZsSg7mY4fXV...

Slot setting

...

Key

Value

Slot setting

...

Connection strings

The connection string values are hidden [Show connection string values](#)

DocDBConnection

< Hidden for Securi...

Custom

Slot setting

...

Name

Value

SQL Database

Slot setting

...

master keys.

- Feel free to continue exploring the Azure Portal, or follow one of our DocumentDB [samples](#) to create your own DocumentDB application.

Next steps

Congratulations! You've deployed DocumentDB, App Service web app and a sample web application using Azure Resource Manager templates.

- To learn more about DocumentDB, click [here](#).
- To learn more about Azure App Service Web apps, click [here](#).
- To learn more about Azure Resource Manager templates, click [here](#).

What's changed

- For a guide to the change from Websites to App Service see: [Azure App Service and Its Impact on Existing Azure Services](#)
- For a guide to the change of the old portal to the new portal see: [Reference for navigating the Azure Classic Portal](#)

NOTE

If you want to get started with Azure App Service before signing up for an Azure account, go to [Try App Service](#), where you can immediately create a short-lived starter web app in App Service. No credit cards required; no commitments.

Logging and error handling in Logic Apps

11/15/2016 • 8 min to read • [Edit on GitHub](#)

Contributors

[Howard S. Edidin](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [Shawn Jackson](#) • [Jeff Hollan](#)

This article describes how you can extend a logic app to better support exception handling. It is a real-life use case and our answer to the question of, "Does Logic Apps support exception and error handling?"

NOTE

The current version of the Logic Apps feature of Microsoft Azure App Service provides a standard template for action responses. This includes both internal validation and error responses returned from an API app.

Overview of the use case and scenario

The following story is the use case for this article. A well-known healthcare organization engaged us to develop an Azure solution that would create a patient portal by using Microsoft Dynamics CRM Online. They needed to send appointment records between the Dynamics CRM Online patient portal and Salesforce. We were asked to use the [HL7 FHIR](#) standard for all patient records.

The project had two major requirements:

- A method to log records sent from the Dynamics CRM Online portal
- A way to view any errors that occurred within the workflow

How we solved the problem

TIP

You can view a high-level video of the project at the [Integration User Group](#).

We chose [Azure DocumentDB](#) as a repository for the log and error records (DocumentDB refers to records as documents). Because Logic Apps has a standard template for all responses, we would not have to create a custom schema. We could create an API app to **Insert** and **Query** for both error and log records. We could also define a schema for each within the API app.

Another requirement was to purge records after a certain date. DocumentDB has a property called [Time to Live](#) (TTL), which allowed us to set a **Time to Live** value for each record or collection. This eliminated the need to manually delete records in DocumentDB.

Creation of the logic app

The first step is to create the logic app and load it in the designer. In this example, we are using parent-child logic apps. Let's assume that we have already created the parent and are going to create one child logic app.

Because we are going to be logging the record coming out of Dynamics CRM Online, let's start at the top. We need to use a Request trigger because the parent logic app triggers this child.

IMPORTANT

To complete this tutorial, you will need to create a DocumentDB database and two collections (Logging and Errors).

Logic app trigger

We are using a Request trigger as shown in the following example.

```
"triggers": {
  "request": {
    "type": "request",
    "kind": "http",
    "inputs": {
      "schema": {
        "properties": {
          "CRMid": {
            "type": "string"
          },
          "recordType": {
            "type": "string"
          },
          "salesforceID": {
            "type": "string"
          },
          "update": {
            "type": "boolean"
          }
        },
        "required": [
          "CRMid",
          "recordType",
          "salesforceID",
          "update"
        ],
        "type": "object"
      }
    }
  }
},
},
```

Steps

We need to log the source (request) of the patient record from the Dynamics CRM Online portal.

1. We need to get a new appointment record from Dynamics CRM Online. The trigger coming from CRM provides us with the **CRM PatientId**, **record type**, **New or Updated Record** (new or update Boolean value), and **SalesforceId**. The **SalesforceId** can be null because it's only used for an update. We will get the CRM record by using the CRM **PatientID** and the **Record Type**.
2. Next, we need to add our DocumentDB API app **InsertLogEntry** operation as shown in the following figures.

Insert log entry designer view

InsertLogEntry

PRESCRIBERID

CRMid

OPERATION

New_Patient

You can insert data from previous steps...

Outputs from manual

BodyCRMidrecordTypesalesforceIDupdate

SOURCE

Headers

SALESFORCEID

salesforceID

DATE

Date

Insert error entry designer view

CreateErrorRecord

Enter a valid integer

STATUSCODE

actions('Create_Appointment')['outputs']['statusCode']

MESSAGE

actions('Create_Appointment')['outputs']['body']['message']

SOURCE

@{concat(triggerBody()['description'],
' START: ', triggerBody()['start'],
' END: ', triggerBody()['end'],
' COMMENT: ', triggerBody()['comment']) }

ACTION

Create_Appointment

ERRORS

RESOLVED

0

NOTES

ISERROR

true

PATIENTID

@{replace(triggerBody()['participant'][0]['actor']['display'],' ','')}

SEVERITY

4

...

Check for create record failure

Condition

CONDITION

@equals(actions('CreateErrorRecord')['status'], 'Failed')

If yes

Add an action

CreateErrorRecord

If no, do nothing

Add an action

Logic app source code

NOTE

The following are samples only. Because this tutorial is based on an implementation currently in production, the value of a **Source Node** might not display properties that are related to scheduling an appointment.

Logging

The following logic app code sample shows how to handle logging.

Log entry

This is the logic app source code for inserting a log entry.

```
"InsertLogEntry": {
  "metadata": {
    "apiDefinitionUrl": "https://.../swagger/docs/v1",
    "swaggerSource": "website"
  },
  "type": "Http",
  "inputs": {
    "body": {
      "date": "@{outputs('Gets_NewPatientRecord')['headers']['Date']}",
      "operation": "New Patient",
      "patientId": "@{triggerBody()['CRMId']}",
      "providerId": "@{triggerBody()['providerID']}",
      "source": "@{outputs('Gets_NewPatientRecord')['headers']}"
    },
    "method": "post",
    "uri": "https://.../api/Log"
  },
  "runAfter": {
    "Gets_NewPatientRecord": ["Succeeded"]
  }
}
```

Log request

This is the log request message posted to the API app.

```
{
  "uri": "https://.../api/Log",
  "method": "post",
  "body": {
    "date": "Fri, 10 Jun 2016 22:31:56 GMT",
    "operation": "New Patient",
    "patientId": "6b115f6d-a7ee-e511-80f5-3863bb2eb2d0",
    "providerId": "",
    "source": "{\n\"Pragma\": \"no-cache\", \"x-ms-request-id\": \"e750c9a9-bd48-44c4-bbba-1688b6f8a132\", \"OData-Version\": \"4.0\", \"Cache-Control\": \"no-cache\", \"Date\": \"Fri, 10 Jun 2016 22:31:56 GMT\", \"Set-Cookie\": \"ARRAffinity=785f4334b5e64d2db0b84edcc1b84f1bf37319679aefce206b51510e56fd9770; Path=/; Domain=127.0.0.1\", \"Server\": \"Microsoft-IIS/8.0, Microsoft-HTTPAPI/2.0\", \"X-AspNet-Version\": \"4.0.30319\", \"X-Powered-By\": \"ASP.NET\", \"Content-Length\": \"1935\", \"Content-Type\": \"application/json; odata.metadata=minimal; odata.streaming=true\", \"Expires\": \"-1\"}"
  }
}
```

Log response

This is the log response message from the API app.

```

{
  "statusCode": 200,
  "headers": {
    "Pragma": "no-cache",
    "Cache-Control": "no-cache",
    "Date": "Fri, 10 Jun 2016 22:32:17 GMT",
    "Server": "Microsoft-IIS/8.0",
    "X-AspNet-Version": "4.0.30319",
    "X-Powered-By": "ASP.NET",
    "Content-Length": "964",
    "Content-Type": "application/json; charset=utf-8",
    "Expires": "-1"
  },
  "body": {
    "ttl": 2592000,
    "id": "6b115f6d-a7ee-e511-80f5-3863bb2eb2d0_1465597937",
    "_rid": "XngRAOT6IQEHAAAAAAAAA==",
    "_self": "dbs/XngRAA==/colls/XngRAOT6IQE=/docs/XngRAOT6IQEHAAAAAAAAA==/",
    "_ts": 1465597936,
    "_etag": "\"0400fc2f-0000-0000-0000-575b3ff00000\"",
    "patientID": "6b115f6d-a7ee-e511-80f5-3863bb2eb2d0",
    "timestamp": "2016-06-10T22:31:56Z",
    "source": "{ \"Pragma\": \"no-cache\", \"x-ms-request-id\": \"e750c9a9-bd48-44c4-bbba-1688b6f8a132\", \"odata-Version\": \"4.0\", \"Cache-Control\": \"no-cache\", \"Date\": \"Fri, 10 Jun 2016 22:31:56 GMT\", \"Set-Cookie\": \"ARRAffinity=785f4334b5e64d2db0b84edcc1b84f1bf37319679aefce206b51510e56fd9770; Path=/; Domain=127.0.0.1\", \"Server\": \"Microsoft-IIS/8.0, Microsoft-HTTPAPI/2.0\", \"X-AspNet-Version\": \"4.0.30319\", \"X-Powered-By\": \"ASP.NET\", \"Content-Length\": \"1935\", \"Content-Type\": \"application/json; odata.metadata=minimal; odata.streaming=true\", \"Expires\": \"-1\" }",
    "operation": "New Patient",
    "salesforceId": "",
    "expired": false
  }
}

```

Now let's look at the error handling steps.

Error handling

The following Logic Apps code sample shows how you can implement error handling.

Create error record

This is the Logic Apps source code for creating an error record.

```

"actions": {
  "CreateErrorRecord": {
    "metadata": {
      "apiDefinitionUrl": "https://.../swagger/docs/v1",
      "swaggerSource": "website"
    },
    "type": "Http",
    "inputs": {
      "body": {
        "action": "New_Patient",
        "isError": true,
        "crmId": "@{triggerBody()['CRMId']}",
        "patientID": "@{triggerBody()['CRMId']}",
        "message": "@{body('Create_NewPatientRecord')['message']}",
        "providerId": "@{triggerBody()['providerId']}",
        "severity": 4,
        "source": "@{actions('Create_NewPatientRecord')['inputs']['body']}",
        "statusCode": "@{int(outputs('Create_NewPatientRecord')['statusCode'])}",
        "salesforceId": "",
        "update": false
      },
      "method": "post",
      "uri": "https://.../api/CrMtoSfError"
    },
    "runAfter": {
      "Create_NewPatientRecord": ["Failed"]
    }
  }
}

```

Insert error into DocumentDB--request

```

{
  "uri": "https://.../api/CrMtoSfError",
  "method": "post",
  "body": {
    "action": "New_Patient",
    "isError": true,
    "crmId": "6b115f6d-a7ee-e511-80f5-3863bb2eb2d0",
    "patientId": "6b115f6d-a7ee-e511-80f5-3863bb2eb2d0",
    "message": "Salesforce failed to complete task: Message: duplicate value found: Account_ID_MED__c
duplicates value on record with id: 001U000001c83gK",
    "providerId": "",
    "severity": 4,
    "salesforceId": "",
    "update": false,
    "source": "
{\"Account_Class_vod__c\": \"PRAC\", \"Account_Status_MED__c\": \"I\", \"CRM_HUB_ID__c\": \"6b115f6d-a7ee-e511-80f5-3863bb2eb2d0\", \"Credentials_vod__c\": \"\", \"DTC_ID_MED__c\": \"\", \"Fax__c\": \"\", \"FirstName__c\": \"A\", \"Gender_vod__c\": \"\", \"IMS_ID__c\": \"\", \"LastName__c\": \"BAILEY\", \"MasterID_mp__c\": \"\", \"C_ID_MED__c\": \"851588\", \"Middle_vod__c\": \"\", \"NPI_vod__c\": \"\", \"PDRP_MED__c\": false, \"PersonDoNotCall__c\": false, \"PersonEmail__c\": \"\", \"PersonHasOptedOutOfEmail__c\": false, \"PersonHasOptedOutOfFax__c\": false, \"PersonMobilePhone__c\": \"\", \"Phone__c\": \"\", \"Practicing_Specialty__c\": \"FM - FAMILY MEDICINE\", \"Primary_City__c\": \"\", \"Primary_State__c\": \"\", \"Primary_Street_Line2__c\": \"\", \"Primary_Street__c\": \"\", \"Primary_Zip__c\": \"\", \"RecordTypeId__c\": \"012U0000000JaPWIA0\", \"Request_Date__c\": \"2016-06-10T22:31:55.9647467Z\", \"ONLY_ID__c\": \"\", \"Specialty_1_vod__c\": \"\", \"Suffix_vod__c\": \"\", \"Website__c\": \"\", \"statusCode\": \"400\"
"
  }
}

```

Insert error into DocumentDB--response

```

{
  "statusCode": 200,
  "headers": {
    "Pragma": "no-cache",
    "Cache-Control": "no-cache",
    "Date": "Fri, 10 Jun 2016 22:31:57 GMT",
    "Server": "Microsoft-IIS/8.0",
    "X-AspNet-Version": "4.0.30319",
    "X-Powered-By": "ASP.NET",
    "Content-Length": "1561",
    "Content-Type": "application/json; charset=utf-8",
    "Expires": "-1"
  },
  "body": {
    "id": "6b115f6d-a7ee-e511-80f5-3863bb2eb2d0-1465597917",
    "_rid": "sQx2APhVzAA8AAAAAAAAA==",
    "_self": "dbs/sQx2AA==/colls/sQx2APhVzAA=/docs/sQx2APhVzAA8AAAAAAAAA==/",
    "_ts": 1465597912,
    "_etag": "\"0c00eaac-0000-0000-0000-575b3fdc0000\"",
    "prescriberId": "6b115f6d-a7ee-e511-80f5-3863bb2eb2d0",
    "timestamp": "2016-06-10T22:31:57.3651027Z",
    "action": "New_Patient",
    "salesforceId": "",
    "update": false,
    "body": "CRM failed to complete task: Message: duplicate value found: CRM_HUB_ID__c duplicates value on record with id: 001U000001c83gK",
    "source": "
{
  \"Account_Class_vod__c\": \"PRAC\",
  \"Account_Status_MED__c\": \"I\",
  \"CRM_HUB_ID__c\": \"6b115f6d-a7ee-e511-80f5-3863bb2eb2d0\",
  \"Credentials_vod__c\": \"DO - Degree level is DO\",
  \"DTC_ID_MED__c\": \"\",
  \"Fax\": \"\",
  \"FirstName\": \"A\",
  \"Gender_vod__c\": \"\",
  \"IMS_ID__c\": \"\",
  \"LastName\": \"BAILEY\",
  \"MterID_mp__c\": \"\",
  \"Medicis_ID_MED__c\": \"851588\",
  \"Middle_vod__c\": \"\",
  \"NPI_vod__c\": \"\",
  \"PDRP_MED__c\": false,
  \"PersonDoNotCall\": false,
  \"PersonEmail\": \"\",
  \"PersonHasOptedOutOfEmail\": false,
  \"PersonHasOptedOutOfFax\": false,
  \"PersonMobilePhone\": \"\",
  \"Phone\": \"\",
  \"Practicing_Specialty__c\": \"FM - FAMILY MEDICINE\",
  \"Primary_City__c\": \"\",
  \"Primary_State__c\": \"\",
  \"Primary_Street_Line2__c\": \"\",
  \"Primary_Street__c\": \"\",
  \"Primary_Zip__c\": \"\",
  \"RecordTypeId\": \"012U0000000JaPWIA0\",
  \"Request_Date__c\": \"2016-06-10T22:31:55.9647467Z\",
  \"XXXXXX\": \"\",
  \"Specialty_1_vod__c\": \"\",
  \"Suffix_vod__c\": \"\",
  \"Website\": \"\"
}
\",
    "code": 400,
    "errors": null,
    "isError": true,
    "severity": 4,
    "notes": null,
    "resolved": 0
  }
}

```

Salesforce error response

```
{
  "statusCode": 400,
  "headers": {
    "Pragma": "no-cache",
    "x-ms-request-id": "3e8e4884-288e-4633-972c-8271b2cc912c",
    "X-Content-Type-Options": "nosniff",
    "Cache-Control": "no-cache",
    "Date": "Fri, 10 Jun 2016 22:31:56 GMT",
    "Set-Cookie":
"ARRAffinity=785f4334b5e64d2db0b84edcc1b84f1bf37319679aefce206b51510e56fd9770;Path=/;Domain=127.0.0.1",
    "Server": "Microsoft-IIS/8.0,Microsoft-HTTPAPI/2.0",
    "X-AspNet-Version": "4.0.30319",
    "X-Powered-By": "ASP.NET",
    "Content-Length": "205",
    "Content-Type": "application/json; charset=utf-8",
    "Expires": "-1"
  },
  "body": {
    "status": 400,
    "message": "Salesforce failed to complete task: Message: duplicate value found: Account_ID_MED__c
duplicates value on record with id: 001U000001c83gK",
    "source": "Salesforce.Common",
    "errors": []
  }
}
```

Returning the response back to the parent logic app

After you have the response, you can pass it back to the parent logic app.

Return success response to the parent logic app

```
"SuccessResponse": {
  "runAfter":
  {
    "UpdateNew_CRMPatientResponse": ["Succeeded"]
  },
  "inputs": {
    "body": {
      "status": "Success"
    }
  },
  "headers": {
    "Content-type": "application/json",
    "x-ms-date": "@utcnow()"
  },
  "statusCode": 200
},
"type": "Response"
}
```

Return error response to the parent logic app

```

"ErrorResponse": {
  "runAfter": {
    {
      "Create_NewPatientRecord": ["Failed"]
    },
  },
  "inputs": {
    "body": {
      "status": "BadRequest"
    },
    "headers": {
      "Content-type": "application/json",
      "x-ms-date": "@utcnow()"
    },
    "statusCode": 400
  },
  "type": "Response"
}

```

DocumentDB repository and portal

Our solution added additional capabilities with [DocumentDB](#).

Error management portal

To view the errors, you can create an MVC web app to display the error records from DocumentDB. **List**, **Details**, **Edit**, and **Delete** operations are included in the current version.

NOTE

Edit operation: DocumentDB does a replace of the entire document. The records shown in the **List** and **Detail** views are samples only. They are not actual patient appointment records.

Following are examples of our MVC app details created with the previously described approach.

Error management list

ivity [Build, Collaborate & ...](#) [Dashboard - Microsoft](#) [Press This](#) [My Account | Wix.com](#) [Press This](#) [bing - Bing](#) [Healthcare Integration](#) [IE](#) [9](#) [+ Paper.li](#) [Add blog to blogroll](#) [Admin D](#)

CRM to SF Error Management

List of Errors

PrescriberId	Action	TimeStamp	Body	Resolved	
ce1820b0-ee2c-e611-80e7-5065f38a5ba1	Create_SFAddress	6/8/2016 4:16:50 PM	Salesforce failed to complete task: Message: duplicate value found: CRM_Hub_Id__c duplicates value on record with id: a01m0000005H3hu	No	Details
ce1820b0-ee2c-e611-80e7-5065f38a5ba1	Create_SFAddress	6/8/2016 4:16:58 PM	Salesforce failed to complete task: Message: duplicate value found: CRM_Hub_Id__c duplicates value on record with id: a01m0000005H3hu	No	Details
4433c660-ac2d-e611-80e7-c4346bdc0271	Create_SFcallPlan	6/8/2016 7:23:00 PM	Salesforce failed to complete task: Message: duplicate value found: External_ID_MED__c duplicates value on record with id: a2pm0000000TFWg	No	Details
12c3f133-b02d-e611-80e7-c4346bdc0271	Update Decile	6/8/2016 7:36:48 PM	Salesforce failed to complete task: Message: Unable to create/update fields: Account_MED__c. Please check the security settings of this field and verify that it is read/write for your profile or permission set.	No	Details
8ea10937-fe2e-e611-80e9-5065f38a3ba1	Update Prescriber	6/10/2016 11:32:30 AM	Salesforce failed to complete task: Message: Provided external ID field does not exist or is not accessible: 001m000000X8lmoAAF123	No	Details
8ea10937-fe2e-e611-80e9-5065f38a3ba1	Update_ATL	6/10/2016 11:32:32 AM	Salesforce failed to complete task: Message: Account: id value of incorrect type: 001m000000X8lmoAAF123	No	Details

Error management detail view

View

Error Response

TimeStamp 6/8/2016 4:16:50 PM
Code 400
Body Salesforce failed to complete task: Message: duplicate value found: CRM_Hub_Id__c duplicates value on record with id: a01m0000005H3hu
Source {"Account_vod__c":"001m000000X74NAAZ","Address_Type_MED__c":"Mailing","Address_line_2_vod__c":"test str 2","CRM_Hub_Id__c":"ce1820b0-ee2c-e611-80e7-5065f38a5ba1","City_vod__c":"edison","Country_vod__c":"United States","Fax_vod__c":"","License_Expiration_Date_vod__c":"2016-06-30","License_State_MED__c":"NJ","License_Status_vod__c":"Valid_vod","License_vod__c":"342198","Name":"test str 1","Phone_vod__c":"","Primary_vod__c":"false","SLX_Address_Line_3__c":"","State_vod__c":"NJ","Zip_vod__c":"435465"}
Action Create_SFAddress
Notes
IsError ☒
PrescriberId ce1820b0-ee2c-e611-80e7-5065f38a5ba1

[Back to List](#)

© 2016 - VNBCConsulting, Inc

Log management portal

To view the logs, we also created an MVC web app. Following are examples of our MVC app details created with the previously described approach.

Sample log detail view

PeterJamesChalmers_1466191912

Document

Save

Discard

Delete

Properties

```

1 {
2   "id": "PeterJamesChalmers_1466191912",
3   "patientId": "PeterJamesChalmers",
4   "timestamp": "2016-06-17T19:31:51.8360138Z",
5   "source": "Discussion on the results of your recent MRI  START: 2013-12-10T09:00:00Z END: 2013-12-10T11:00:00Z COMMENT: Further expand on the results of the MRI and determine the next actions that may be appropriate.",
6   "operation": "General Discussion",
7   "Provider": "DrAdamCareful",
8   "ttl": 2592000,
9   "notDeleted": true
10 }
```

Properties

PeterJamesChalmers_1466191912

_RID

Uw4fAJrEEwEqAAAAAAAAAA==

_TS

Fri, 17 Jun 2016 19:31:50 GMT

_SELF

dbs/Uw4fAA==/colls/Uw4fAJrEEwE=/doc

_ETAG

"0000dc00-0000-0000-0000-576450290C"

_ATTACHMENTS

attachments/

API app details

Logic Apps exception management API

Our open-source Logic Apps exception management API app provides the following functionality.

There are two controllers:

- **ErrorController** inserts an error record (document) in a DocumentDB collection.
- **LogController** Inserts a log record (document) in a DocumentDB collection.

TIP

Both controllers use `async Task<dynamic>` operations. This allows operations to be resolved at runtime, so we can create the DocumentDB schema in the body of the operation.

Every document in DocumentDB must have a unique ID. We are using `PatientId` and adding a timestamp that is converted to a Unix timestamp value (double). We truncate it to remove the fractional value.

You can view the source code of our error controller API [from GitHub](#).

We call the API from a logic app by using the following syntax.

```
"actions": {
  "CreateErrorRecord": {
    "metadata": {
      "apiDefinitionUrl": "https://.../swagger/docs/v1",
      "swaggerSource": "website"
    },
    "type": "Http",
    "inputs": {
      "body": {
        "action": "New_Patient",
        "isError": true,
        "crmId": "@{triggerBody()['CRMId']}",
        "prescriberId": "@{triggerBody()['CRMId']}",
        "message": "@{body('Create_NewPatientRecord')['message']}",
        "salesforceId": "@{triggerBody()['salesforceID']}",
        "severity": 4,
        "source": "@{actions('Create_NewPatientRecord')['inputs']['body']}",
        "statusCode": "@{int(outputs('Create_NewPatientRecord')['statusCode'])}",
        "update": false
      },
      "method": "post",
      "uri": "https://.../api/CrMtoSfError"
    },
    "runAfter": {
      "Create_NewPatientRecord": ["Failed"]
    }
  }
}
```

The expression in the preceding code sample is checking for the *Create_NewPatientRecord* status of **Failed**.

Summary

- You can easily implement logging and error handling in a logic app.
- You can use DocumentDB as the repository for log and error records (documents).
- You can use MVC to create a portal to display log and error records.

Source code

The source code for the Logic Apps exception management API application is available in this [GitHub repository](#).

Next steps

- [View more Logic Apps examples and scenarios](#)
- [Learn about Logic Apps monitoring tools](#)
- [Create a Logic App automated deployment template](#)

Azure Functions DocumentDB bindings

11/15/2016 • 5 min to read • [Edit on GitHub](#)

Contributors

[Christopher Anderson](#) • [wesmc](#) • [Glenn Gailey](#) • [cephalin](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [Sylvan Clebsch](#) • [Tom Dykstra](#)

This article explains how to configure and code Azure DocumentDB bindings in Azure Functions. Azure Functions supports input and output bindings for DocumentDB.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- [Create your first Azure Function](#)
- [Azure Functions developer reference](#)
- [C#, F#, or Node developer reference](#)

For more information on DocumentDB, see [Introduction to DocumentDB](#) and [Build a DocumentDB console application](#).

DocumentDB input binding

The DocumentDB input binding retrieves a DocumentDB document and passes it to the named input parameter of the function. The document ID can be determined based on the trigger that invokes the function.

The DocumentDB input to a function uses the following JSON object in the `bindings` array of `function.json`:

```
{
  "name": "<Name of input parameter in function signature>",
  "type": "documentDB",
  "databaseName": "<Name of the DocumentDB database>",
  "collectionName": "<Name of the DocumentDB collection>",
  "id": "<Id of the DocumentDB document - see below>",
  "connection": "<Name of app setting with connection string - see below>",
  "direction": "in"
},
```

Note the following:

- `id` supports bindings similar to `{queueTrigger}`, which uses the string value of the queue message as the document ID.
- `connection` must be the name of an app setting that points to the endpoint for your DocumentDB account (with the value `AccountEndpoint=<Endpoint for your account>;AccountKey=<Your primary access key>`). If you create a DocumentDB account through the Functions portal UI, the account creation process creates an app setting for you. To use an existing DocumentDB account, you need to [configure this app setting manually]().
- If the specified document is not found, the named input parameter to the function is set to `null`.

Input usage

This section shows you how to use your DocumentDB input binding in your function code.

In C# and F# functions, any changes made to the input document (named input parameter) is automatically sent

back to the collection when the function exits successfully. In Node.js functions, updates to the document in the input binding are not sent back to the collection. However, you can use `context.bindings.<documentName>In` and `context.bindings.<documentName>Out` to make updates to input documents. See how it is done in the [Node.js sample](#).

Input sample

Suppose you have the following DocumentDB input binding in the `bindings` array of `function.json`:

```
{
  "name": "inputDocument",
  "type": "documentDB",
  "databaseName": "MyDatabase",
  "collectionName": "MyCollection",
  "id" : "{queueTrigger}",
  "connection": "MyAccount_DOCUMENTDB",
  "direction": "in"
}
```

See the language-specific sample that uses this input binding to update the document's text value.

- [C#](#)
- [F#](#)
- [Node.js](#)

Input sample in C#

```
public static void Run(string myQueueItem, dynamic inputDocument)
{
    inputDocument.text = "This has changed.";
}
```

Input sample in F#

```
open FSharp.Interop.Dynamic
let Run(myQueueItem: string, inputDocument: obj) =
    inputDocument?text <- "This has changed."
```

You need to add a `project.json` file that specifies the `FSharp.Interop.Dynamic` and `Dynamitey` NuGet dependencies:

```
{
  "frameworks": {
    "net46": {
      "dependencies": {
        "Dynamitey": "1.0.2",
        "FSharp.Interop.Dynamic": "3.0.0"
      }
    }
  }
}
```

To add a `project.json` file, see [F# package management](#).

Input sample in Node.js

```
module.exports = function (context) {  
  context.bindings.inputDocumentOut = context.bindings.inputDocumentIn;  
  context.bindings.inputDocumentOut.text = "This was updated!";  
  context.done();  
};
```

DocumentDB output binding

The DocumentDB output binding lets you write a new document to an Azure DocumentDB database.

The output binding uses the following JSON object in the `bindings` array of `function.json`:

```
{  
  "name": "<Name of output parameter in function signature>",  
  "type": "documentDB",  
  "databaseName": "<Name of the DocumentDB database>",  
  "collectionName": "<Name of the DocumentDB collection>",  
  "createIfNotExists": <true or false - see below>,  
  "connection": "<Value of AccountEndpoint in Application Setting - see below>",  
  "direction": "out"  
}
```

Note the following:

- Set `createIfNotExists` to `true` to create the database and collection if it doesn't exist. The default value is `false`. New collections are created with reserved throughput, which has pricing implications. For more information, see [DocumentDB pricing](#).
- `connection` must be the name of an app setting that points to the endpoint for your DocumentDB account (with the value `AccountEndpoint=<Endpoint for your account>;AccountKey=<Your primary access key>`). If you create a DocumentDB account through the Functions portal UI, the account creation process creates a new app setting for you. To use an existing DocumentDB account, you need to [configure this app setting manually]().

Output usage

This section shows you how to use your DocumentDB output binding in your function code.

When you write to the output parameter in your function, by default a new document is generated in your database, with an automatically generated GUID as the document ID. You can specify the document ID of output document by specifying the `id` JSON property in the output parameter. If a document with that ID already exists, the output document overwrites it.

Output sample

Suppose you have the following DocumentDB output binding in the `bindings` array of `function.json`:

```
{  
  "name": "employeeDocument",  
  "type": "documentDB",  
  "databaseName": "MyDatabase",  
  "collectionName": "MyCollection",  
  "createIfNotExists": true,  
  "connection": "MyAccount_DOCUMENTDB",  
  "direction": "out"  
}
```

And you have a queue input binding for a queue that receives JSON in the following format:

```
{
  "name": "John Henry",
  "employeeId": "123456",
  "address": "A town nearby"
}
```

And you want to create DocumentDB documents in the following format for each record:

```
{
  "id": "John Henry-123456",
  "name": "John Henry",
  "employeeId": "123456",
  "address": "A town nearby"
}
```

See the language-specific sample that uses this output binding to add documents to your database.

- [C#](#)
- [F#](#)
- [Node.js](#)

Output sample in C#

```
#r "Newtonsoft.Json"

using System;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;

public static void Run(string myQueueItem, out object employeeDocument, TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: {myQueueItem}");

    dynamic employee = JObject.Parse(myQueueItem);

    employeeDocument = new {
        id = employee.name + "-" + employee.employeeId,
        name = employee.name,
        employeeId = employee.employeeId,
        address = employee.address
    };
}
```

Output sample in F#

```

open FSharp.Interop.Dynamic
open Newtonsoft.Json

type Employee = {
    id: string
    name: string
    employeeId: string
    address: string
}

let Run(myQueueItem: string, employeeDocument: byref<obj>, log: TraceWriter) =
    log.Info(sprintf "F# Queue trigger function processed: %s" myQueueItem)
    let employee = JObject.Parse(myQueueItem)
    employeeDocument <-
        { id = sprintf "%s-%s" employee?name employee?employeeId
          name = employee?name
          employeeId = employee?employeeId
          address = employee?address }

```

You need to add a `project.json` file that specifies the `FSharp.Interop.Dynamic` and `Dynamitey` NuGet dependencies:

```

{
  "frameworks": {
    "net46": {
      "dependencies": {
        "Dynamitey": "1.0.2",
        "FSharp.Interop.Dynamic": "3.0.0"
      }
    }
  }
}

```

To add a `project.json` file, see [F# package management](#).

Output sample in Node.js

```

module.exports = function (context) {

    context.bindings.employeeDocument = JSON.stringify({
        id: context.bindings.myQueueItem.name + "-" + context.bindings.myQueueItem.employeeId,
        name: context.bindings.myQueueItem.name,
        employeeId: context.bindings.myQueueItem.employeeId,
        address: context.bindings.myQueueItem.address
    });

    context.done();
};

```

Run a Hadoop job using DocumentDB and HDInsight

11/15/2016 • 15 min to read • [Edit on GitHub](#)

Contributors

[Denny Lee](#) • [Theano Petersen](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [mimig](#) • [Andrew Hoh](#) • [nitinme](#) • [v-aljenk](#)

This tutorial shows you how to run [Apache Hive](#), [Apache Pig](#), and [Apache Hadoop](#) MapReduce jobs on Azure HDInsight with DocumentDB's Hadoop connector. DocumentDB's Hadoop connector allows DocumentDB to act as both a source and sink for Hive, Pig, and MapReduce jobs. This tutorial will use DocumentDB as both the data source and destination for Hadoop jobs.

After completing this tutorial, you'll be able to answer the following questions:

- How do I load data from DocumentDB using a Hive, Pig, or MapReduce job?
- How do I store data in DocumentDB using a Hive, Pig, or MapReduce job?

We recommend getting started by watching the following video, where we run through a Hive job using DocumentDB and HDInsight.

Then, return to this article, where you'll receive the full details on how you can run analytics jobs on your DocumentDB data.

TIP

This tutorial assumes that you have prior experience using Apache Hadoop, Hive, and/or Pig. If you are new to Apache Hadoop, Hive, and Pig, we recommend visiting the [Apache Hadoop documentation](#). This tutorial also assumes that you have prior experience with DocumentDB and have a DocumentDB account. If you are new to DocumentDB or you do not have a DocumentDB account, please check out our [Getting Started](#) page.

Don't have time to complete the tutorial and just want to get the full sample PowerShell scripts for Hive, Pig, and MapReduce? Not a problem, get them [here](#). The download also contains the hql, pig, and java files for these

samples.

Newest Version

HADOOP CONNECTOR VERSION	1.2.0
SCRIPT URI	https://portalcontent.blob.core.windows.net/scriptaction/documentdb-hadoop-installer-v04.ps1
DATE MODIFIED	04/26/2016
SUPPORTED HDINSIGHT VERSIONS	3.1, 3.2
CHANGE LOG	Updated DocumentDB Java SDK to 1.6.0 Added support for partitioned collections as both a source and sink

Prerequisites

Before following the instructions in this tutorial, ensure that you have the following:

- A DocumentDB account, a database, and a collection with documents inside. For more information, see [Getting Started with DocumentDB](#). Import sample data into your DocumentDB account with the [DocumentDB import tool](#).
- Throughput. Reads and writes from HDInsight will be counted towards your allotted request units for your collections. For more information, see [Provisioned throughput, request units, and database operations](#).
- Capacity for an additional stored procedure within each output collection. The stored procedures are used for transferring resulting documents. For more information, see [Collections and provisioned throughput](#).
- Capacity for the resulting documents from the Hive, Pig, or MapReduce jobs. For more information, see [Manage DocumentDB capacity and performance](#).
- *[Optional]* Capacity for an additional collection. For more information, see [Provisioned document storage and index overhead](#).

WARNING

In order to avoid the creation of a new collection during any of the jobs, you can either print the results to stdout, save the output to your WASB container, or specify an already existing collection. In the case of specifying an existing collection, new documents will be created inside the collection and already existing documents will only be affected if there is a conflict in *ids*. **The connector will automatically overwrite existing documents with id conflicts.** You can turn off this feature by setting the upsert option to false. If upsert is false and a conflict occurs, the Hadoop job will fail; reporting an id conflict error.

Step 1: Create a new HDInsight cluster

This tutorial uses Script Action from the Azure Portal to customize your HDInsight cluster. In this tutorial, we will use the Azure Portal to create your HDInsight cluster. For instructions on how to use PowerShell cmdlets or the HDInsight .NET SDK, check out the [Customize HDInsight clusters using Script Action](#) article.

1. Sign in to the [Azure Portal](#).
2. Click + **New** on the top of the left navigation, search for **HDInsight** in the top search bar on the New blade.
3. **HDInsight** published by **Microsoft** will appear at the top of the Results. Click on it and then click **Create**.
4. On the New HDInsight Cluster create blade, enter your **Cluster Name** and select the **Subscription** you want to provision this resource under.

Cluster name	Name the cluster. DNS name must start and end with an alpha numeric character, and may contain dashes. The field must be a string between 3 and 63 characters.
Subscription Name	If you have more than one Azure Subscription, select the subscription that will host your HDInsight cluster.

5. Click **Select Cluster Type** and set the following properties to the specified values.

Cluster type	Hadoop
Cluster tier	Standard
Operating System	Windows
Version	latest version

Now, click **SELECT**.

The screenshot shows the 'New HDInsight Cluster' blade in the Azure portal, specifically the 'Cluster Type configuration' tab. The left sidebar contains several sections: 'Cluster Name' (set to 'AzureDocumentDB'), 'Subscription' (set to 'The Data Platform'), 'Select Cluster Type' (highlighted with a red box and a red arrow), 'Applications', 'Credentials', 'Data Source', 'Pricing', 'Optional Configuration', and 'Resource Group'. The main area displays the configuration for the 'Cluster Type'. It shows 'Cluster Type' as 'Hadoop', 'Operating System' as 'Windows', and 'Version' as 'Hadoop 2.7.0 (HDI 3.3)'. Below this, the 'Cluster Tier' is shown with two options: 'STANDARD' and 'PREMIUM (PREVIEW)'. The 'STANDARD' tier is selected, showing a price of '+ 0.00 USD/CORE/HOUR'. The 'PREMIUM (PREVIEW)' tier shows a price of '+ 0.02 USD/CORE/HOUR'. A 'Select' button is located at the bottom right of the main area.

6. Click on **Credentials** to set your login and remote access credentials. Choose your **Cluster Login Username** and **Cluster Login Password**.

If you want to remote into your cluster, select *yes* at the bottom of the blade and provide a username and password.

- Click on **Data Source** to set your primary location for data access. Choose the **Selection Method** and specify an already existing storage account or create a new one.
- On the same blade, specify a **Default Container** and a **Location**. And, click **SELECT**.

NOTE

Select a location close to your DocumentDB account region for better performance

- Click on **Pricing** to select the number and type of nodes. You can keep the default configuration and scale the number of Worker nodes later on.
- Click **Optional Configuration**, then **Script Actions** in the Optional Configuration Blade.

In Script Actions, enter the following information to customize your HDInsight cluster.

PROPERTY	VALUE
Name	Specify a name for the script action.
Script URI	Specify the URI to the script that is invoked to customize the cluster. Please enter: https://portalcontent.blob.core.windows.net/scriptaction/documentdb-hadoop-installer-v04.ps1 .
Head	Click the checkbox to run the PowerShell script onto the Head node. Check this checkbox.
Worker	Click the checkbox to run the PowerShell script onto the Worker node. Check this checkbox.
Zookeeper	Click the checkbox to run the PowerShell script onto the Zookeeper. Not needed.
Parameters	Specify the parameters, if required by the script. No Parameters needed.

- Create either a new **Resource Group** or use an existing Resource Group under your Azure Subscription.
- Now, check **Pin to dashboard** to track its deployment and click **Create!**

Step 2: Install and configure Azure PowerShell

- Install Azure PowerShell. Instructions can be found [here](#).

NOTE

Alternatively, just for Hive queries, you can use HDInsight's online Hive Editor. To do so, sign in to the [Azure Portal](#), click **HDInsight** on the left pane to view a list of your HDInsight clusters. Click the cluster you want to run Hive queries on, and then click **Query Console**.

- Open the Azure PowerShell Integrated Scripting Environment:
 - On a computer running Windows 8 or Windows Server 2012 or higher, you can use the built-in Search.

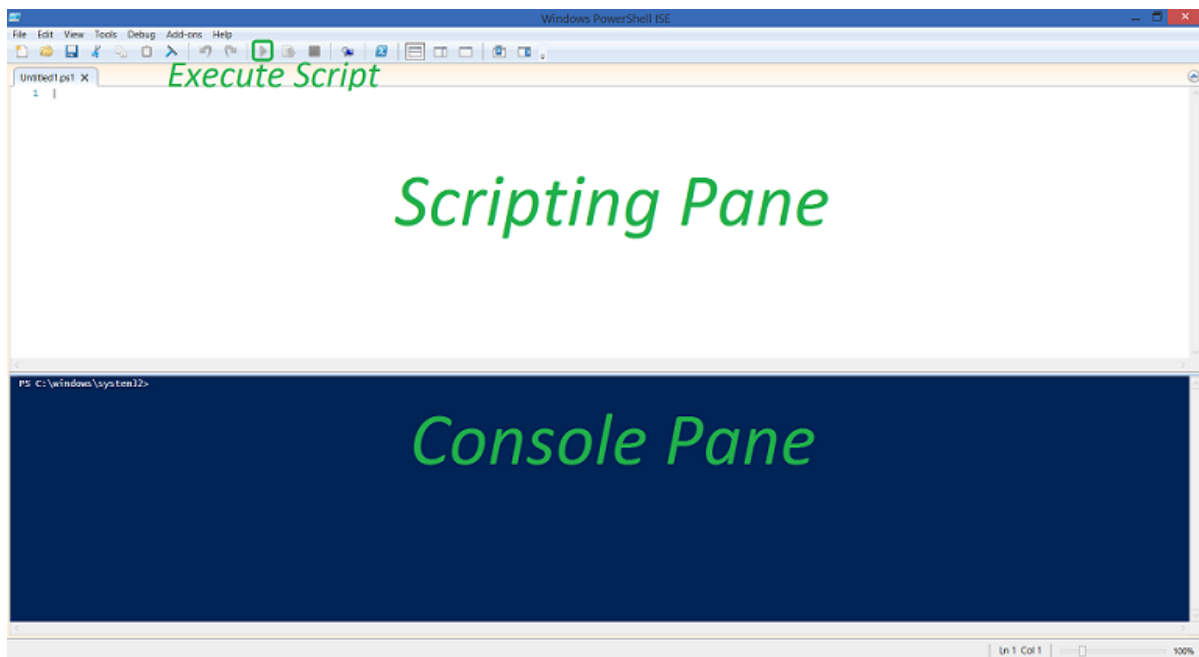
From the Start screen, type **powershell ise** and click **Enter**.

- On a computer running a version earlier than Windows 8 or Windows Server 2012, use the Start menu. From the Start menu, type **Command Prompt** in the search box, then in the list of results, click **Command Prompt**. In the Command Prompt, type **powershell_ise** and click **Enter**.

3. Add your Azure Account.

- a. In the Console Pane, type **Add-AzureAccount** and click **Enter**.
- b. Type in the email address associated with your Azure subscription and click **Continue**.
- c. Type in the password for your Azure subscription.
- d. Click **Sign in**.

4. The following diagram identifies the important parts of your Azure PowerShell Scripting Environment.



Step 3: Run a Hive job using DocumentDB and HDInsight

IMPORTANT

All variables indicated by < > must be filled in using your configuration settings.

1. Set the following variables in your PowerShell Script pane.

```
# Provide Azure subscription name, the Azure Storage account and container that is used for the default
HDInsight file system.
$subscriptionName = "<SubscriptionName>"
$storageAccountName = "<AzureStorageAccountName>"
$containerName = "<AzureStorageContainerName>"

# Provide the HDInsight cluster name where you want to run the Hive job.
$clusterName = "<HDInsightClusterName>"
```

2. Let's begin constructing your query string. We'll write a Hive query that takes all documents' system generated timestamps (`_ts`) and unique ids (`_rid`) from a DocumentDB collection, tallies all documents by the minute, and then stores the results back into a new DocumentDB collection.

First, let's create a Hive table from our DocumentDB collection. Add the following code snippet to the PowerShell Script pane **after** the code snippet from #1. Make sure you include the optional DocumentDB.query parameter to trim our documents to just `_ts` and `_rid`.

NOTE

Naming `DocumentDB.inputCollections` was not a mistake. Yes, we allow adding multiple collections as an input:

```
'*DocumentDB.inputCollections*' = '*\<DocumentDB Input Collection Name 1\>*,*\<DocumentDB Input
Collection Name 2\>*' A1A</br> The collection names are separated without spaces, using only a single
comma.

# Create a Hive table using data from DocumentDB. Pass DocumentDB the query to filter transferred data to
_rid and _ts.
$queryStringPart1 = "drop table DocumentDB_timestamps; " +
    "create external table DocumentDB_timestamps(id string, ts BIGINT) " +
    "stored by 'com.microsoft.azure.documentdb.hive.DocumentDBStorageHandler' " +
    "tblproperties ( " +
        "'DocumentDB.endpoint' = '<DocumentDB Endpoint>', " +
        "'DocumentDB.key' = '<DocumentDB Primary Key>', " +
        "'DocumentDB.db' = '<DocumentDB Database Name>', " +
        "'DocumentDB.inputCollections' = '<DocumentDB Input Collection Name>', " +
        "'DocumentDB.query' = 'SELECT r._rid AS id, r._ts AS ts FROM root r' ); "
```

- Next, let's create a Hive table for the output collection. The output document properties will be the month, day, hour, minute, and the total number of occurrences.

NOTE

Yet again, naming `DocumentDB.outputCollections` was not a mistake. Yes, we allow adding multiple collections as an output:

```
'DocumentDB.outputCollections' = '<DocumentDB Output Collection Name 1>,<DocumentDB Output Collection
Name 2>'
```

The collection names are separated without spaces, using only a single comma.

Documents will be distributed round-robin across multiple collections. A batch of documents will be stored in one collection, then a second batch of documents will be stored in the next collection, and so forth.

```
# Create a Hive table for the output data to DocumentDB.
$queryStringPart2 = "drop table DocumentDB_analytics; " +
    "create external table DocumentDB_analytics(Month INT, Day INT, Hour INT, Minute
INT, Total INT) " +
    "stored by 'com.microsoft.azure.documentdb.hive.DocumentDBStorageHandler' " +
    "tblproperties ( " +
        "'DocumentDB.endpoint' = '<DocumentDB Endpoint>', " +
        "'DocumentDB.key' = '<DocumentDB Primary Key>', " +
        "'DocumentDB.db' = '<DocumentDB Database Name>', " +
        "'DocumentDB.outputCollections' = '<DocumentDB Output Collection Name>' ); "
```

- Finally, let's tally the documents by month, day, hour, and minute and insert the results back into the output Hive table.

```
# GROUP BY minute, COUNT entries for each, INSERT INTO output Hive table.
$queryStringPart3 = "INSERT INTO table DocumentDB_analytics " +
    "SELECT month(from_unixtime(ts)) as Month, day(from_unixtime(ts)) as Day, " +
    "hour(from_unixtime(ts)) as Hour, minute(from_unixtime(ts)) as Minute, " +
    "COUNT(*) AS Total " +
    "FROM DocumentDB_timestamps " +
    "GROUP BY month(from_unixtime(ts)), day(from_unixtime(ts)), " +
    "hour(from_unixtime(ts)) , minute(from_unixtime(ts)); "
```

- Add the following script snippet to create a Hive job definition from the previous query.

```
# Create a Hive job definition.
$queryString = $queryStringPart1 + $queryStringPart2 + $queryStringPart3
$hiveJobDefinition = New-AzureHDInsightHiveJobDefinition -Query $queryString
```

You can also use the -File switch to specify a HiveQL script file on HDFS.

6. Add the following snippet to save the start time and submit the Hive job.

```
# Save the start time and submit the job to the cluster.
$startTime = Get-Date
Select-AzureSubscription $subscriptionName
$hiveJob = Start-AzureHDInsightJob -Cluster $clusterName -JobDefinition $hiveJobDefinition
```

7. Add the following to wait for the Hive job to complete.

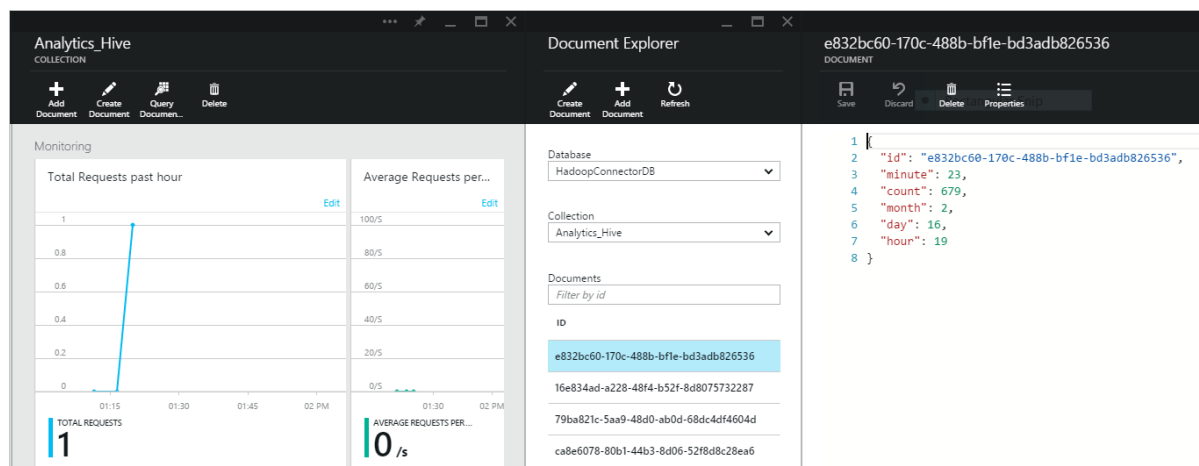
```
# Wait for the Hive job to complete.
Wait-AzureHDInsightJob -Job $hiveJob -WaitTimeoutInSeconds 3600
```

8. Add the following to print the standard output and the start and end times.

```
# Print the standard error, the standard output of the Hive job, and the start and end time.
$endTime = Get-Date
Get-AzureHDInsightJobOutput -Cluster $clusterName -JobId $hiveJob.JobId -StandardOutput
Write-Host "Start: " $startTime ", End: " $endTime -ForegroundColor Green
```

9. Run your new script! Click the green execute button.
10. Check the results. Sign into the [Azure Portal](#).
 - a. Click **Browse** on the left-side panel.
 - b. Click **everything** at the top-right of the browse panel.
 - c. Find and click **DocumentDB Accounts**.
 - d. Next, find your **DocumentDB Account**, then **DocumentDB Database** and your **DocumentDB Collection** associated with the output collection specified in your Hive query.
 - e. Finally, click **Document Explorer** underneath **Developer Tools**.

You will see the results of your Hive query.



Step 4: Run a Pig job using DocumentDB and HDInsight

IMPORTANT

All variables indicated by < > must be filled in using your configuration settings.

1. Set the following variables in your PowerShell Script pane.

```
# Provide Azure subscription name.
$subscriptionName = "Azure Subscription Name"

# Provide HDInsight cluster name where you want to run the Pig job.
$clusterName = "Azure HDInsight Cluster Name"
```

2. Let's begin constructing your query string. We'll write a Pig query that takes all documents' system generated timestamps (_ts) and unique ids (_rid) from a DocumentDB collection, tallies all documents by the minute, and then stores the results back into a new DocumentDB collection.

First, load documents from DocumentDB into HDInsight. Add the following code snippet to the PowerShell Script pane **after** the code snippet from #1. Make sure to add a DocumentDB query to the optional DocumentDB query parameter to trim our documents to just _ts and _rid.

NOTE

Yes, we allow adding multiple collections as an input:

'<DocumentDB Input Collection Name 1>,<DocumentDB Input Collection Name 2>'

The collection names are separated without spaces, using only a single comma.

Documents will be distributed round-robin across multiple collections. A batch of documents will be stored in one collection, then a second batch of documents will be stored in the next collection, and so forth.

```
# Load data from DocumentDB. Pass DocumentDB query to filter transferred data to _rid and _ts.
$queryStringPart1 = "DocumentDB_timestamps = LOAD '<DocumentDB Endpoint>' USING
com.microsoft.azure.documentdb.pig.DocumentDBLoader( " +
    "'<DocumentDB Primary Key>', " +
    "'<DocumentDB Database Name>', " +
    "'<DocumentDB Input Collection Name>', " +
    "'SELECT r._rid AS id, r._ts AS ts FROM root r' ); "
```

3. Next, let's tally the documents by the month, day, hour, minute, and the total number of occurrences.

```
# GROUP BY minute and COUNT entries for each.
$queryStringPart2 = "timestamp_record = FOREACH DocumentDB_timestamps GENERATE `${0}#id' as id:int,
ToDate((long)(`${0}#ts') * 1000) as timestamp:datetime; " +
    "by_minute = GROUP timestamp_record BY (GetYear(timestamp), GetMonth(timestamp),
GetDay(timestamp), GetHour(timestamp), GetMinute(timestamp)); " +
    "by_minute_count = FOREACH by_minute GENERATE FLATTEN(group) as (Year:int, Month:int,
Day:int, Hour:int, Minute:int), COUNT(timestamp_record) as Total:int; "
```

4. Finally, let's store the results into our new output collection.

NOTE

Yes, we allow adding multiple collections as an output:

'<DocumentDB Output Collection Name 1>,<DocumentDB Output Collection Name 2>'

The collection names are separated without spaces, using only a single comma.

Documents will be distributed round-robin across the multiple collections. A batch of documents will be stored in one collection, then a second batch of documents will be stored in the next collection, and so forth.

```
# Store output data to DocumentDB.
$queryStringPart3 = "STORE by_minute_count INTO '<DocumentDB Endpoint>' " +
    "USING com.microsoft.azure.documentdb.pig.DocumentDBStorage( " +
    "'<DocumentDB Primary Key>', " +
    "'<DocumentDB Database Name>', " +
    "'<DocumentDB Output Collection Name>'); "
```

5. Add the following script snippet to create a Pig job definition from the previous query.

```
# Create a Pig job definition.
$queryString = $queryStringPart1 + $queryStringPart2 + $queryStringPart3
$pigJobDefinition = New-AzureHDInsightPigJobDefinition -Query $queryString -StatusFolder $statusFolder
```

You can also use the -File switch to specify a Pig script file on HDFS.

6. Add the following snippet to save the start time and submit the Pig job.

```
# Save the start time and submit the job to the cluster.
$startTime = Get-Date
Select-AzureSubscription $subscriptionName
$pigJob = Start-AzureHDInsightJob -Cluster $clusterName -JobDefinition $pigJobDefinition
```

7. Add the following to wait for the Pig job to complete.

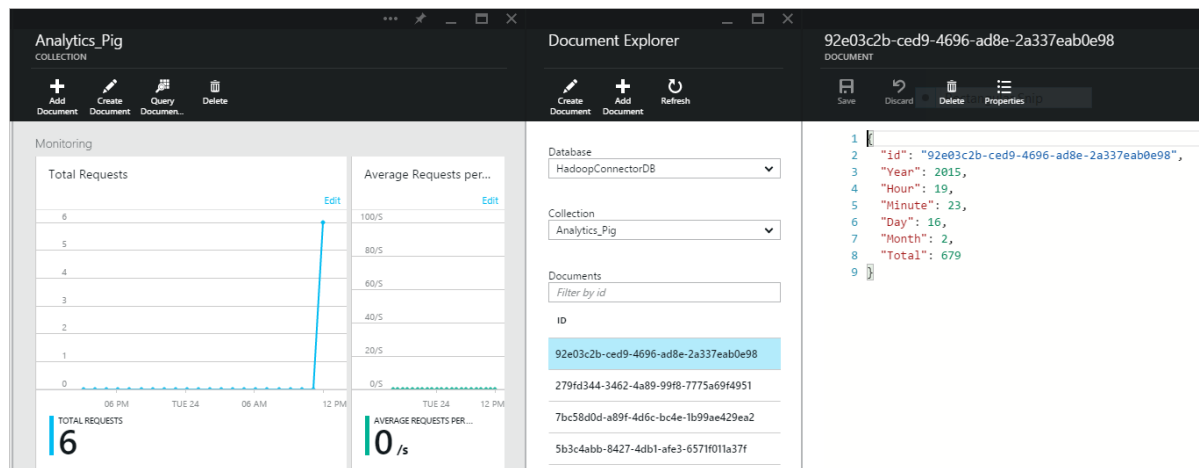
```
# Wait for the Pig job to complete.
Wait-AzureHDInsightJob -Job $pigJob -WaitTimeoutInSeconds 3600
```

8. Add the following to print the standard output and the start and end times.

```
# Print the standard error, the standard output of the Hive job, and the start and end time.
$endTime = Get-Date
Get-AzureHDInsightJobOutput -Cluster $clusterName -JobId $pigJob.JobId -StandardOutput
Write-Host "Start: " $startTime ", End: " $endTime -ForegroundColor Green
```

9. **Run** your new script! Click the green execute button.
10. Check the results. Sign into the [Azure Portal](#).
 - a. Click **Browse** on the left-side panel.
 - b. Click **everything** at the top-right of the browse panel.
 - c. Find and click **DocumentDB Accounts**.
 - d. Next, find your **DocumentDB Account**, then **DocumentDB Database** and your **DocumentDB Collection** associated with the output collection specified in your Pig query.
 - e. Finally, click **Document Explorer** underneath **Developer Tools**.

You will see the results of your Pig query.



Step 5: Run a MapReduce job using DocumentDB and HDInsight

1. Set the following variables in your PowerShell Script pane.

```
$subscriptionName = "<SubscriptionName>" # Azure subscription name
$clusterName = "<ClusterName>" # HDInsight cluster name
```

2. We'll execute a MapReduce job that tallies the number of occurrences for each Document property from your DocumentDB collection. Add this script snippet **after** the snippet above.

```
# Define the MapReduce job.
$TallyPropertiesJobDefinition = New-AzureHDInsightMapReduceJobDefinition -JarFile
"wasb:///example/jars/TallyProperties-v01.jar" -ClassName "TallyProperties" -Arguments "<DocumentDB
Endpoint>","<DocumentDB Primary Key>","<DocumentDB Database Name>","<DocumentDB Input Collection Name>","
<DocumentDB Output Collection Name>","<[Optional] DocumentDB Query>"
```

NOTE

TallyProperties-v01.jar comes with the custom installation of the DocumentDB Hadoop Connector.

3. Add the following command to submit the MapReduce job.

```
# Save the start time and submit the job.
$startTime = Get-Date
Select-AzureSubscription $subscriptionName
$TallyPropertiesJob = Start-AzureHDInsightJob -Cluster $clusterName -JobDefinition
$TallyPropertiesJobDefinition | Wait-AzureHDInsightJob -WaitTimeoutInSeconds 3600
```

In addition to the MapReduce job definition, you also provide the HDInsight cluster name where you want to run the MapReduce job, and the credentials. The `Start-AzureHDInsightJob` is an asynchronous call. To check the completion of the job, use the `Wait-AzureHDInsightJob` cmdlet.

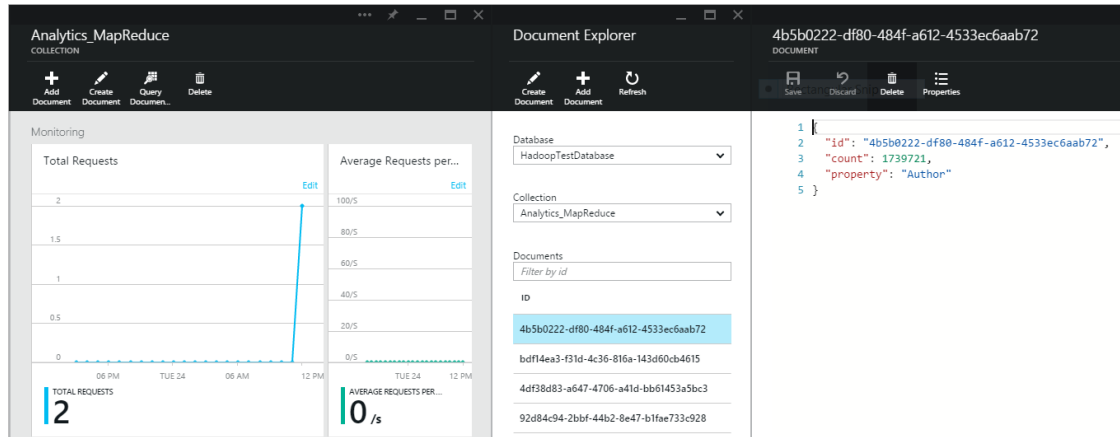
4. Add the following command to check any errors with running the MapReduce job.

```
# Get the job output and print the start and end time.
$endTime = Get-Date
Get-AzureHDInsightJobOutput -Cluster $clusterName -JobId $TallyPropertiesJob.JobId -StandardError
Write-Host "Start: " $startTime ", End: " $endTime -ForegroundColor Green
```

5. Run your new script! Click the green execute button.
6. Check the results. Sign into the [Azure Portal](#).

- Click **Browse** on the left-side panel.
- Click **everything** at the top-right of the browse panel.
- Find and click **DocumentDB Accounts**.
- Next, find your **DocumentDB Account**, then **DocumentDB Database** and your **DocumentDB Collection** associated with the output collection specified in your MapReduce job.
- Finally, click **Document Explorer** underneath **Developer Tools**.

You will see the results of your MapReduce job.



Next Steps

Congratulations! You just ran your first Hive, Pig, and MapReduce jobs using Azure DocumentDB and HDInsight.

We have open sourced our Hadoop Connector. If you're interested, you can contribute on [GitHub](#).

To learn more, see the following articles:

- [Develop a Java application with Documentdb](#)
- [Develop Java MapReduce programs for Hadoop in HDInsight](#)
- [Get started using Hadoop with Hive in HDInsight to analyze mobile handset use](#)
- [Use MapReduce with HDInsight](#)
- [Use Hive with HDInsight](#)
- [Use Pig with HDInsight](#)
- [Customize HDInsight clusters using Script Action](#)

Connecting DocumentDB with Azure Search using indexers

11/15/2016 • 7 min to read • [Edit on GitHub](#)

Contributors

Denny Lee • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • mimig • Andrew Hoh • v-aljenk • Andrew Liu
• Dene Hager

If you're looking to implement great search experiences over your DocumentDB data, use Azure Search indexer for DocumentDB! In this article, we will show you how to integrate Azure DocumentDB with Azure Search without having to write any code to maintain indexing infrastructure!

To set this up, you have to [setup an Azure Search account](#) (you don't need to upgrade to standard search), and then call the [Azure Search REST API](#) to create a DocumentDB **data source** and an **indexer** for that data source.

In order send requests to interact with the REST APIs, you can use [Postman](#), [Fiddler](#), or any tool of your preference.

Azure Search indexer concepts

Azure Search supports the creation and management of data sources (including DocumentDB) and indexers that operate against those data sources.

A **data source** specifies what data needs to be indexed, credentials to access the data, and policies to enable Azure Search to efficiently identify changes in the data (such as modified or deleted documents inside your collection). The data source is defined as an independent resource so that it can be used by multiple indexers.

An **indexer** describes how the data flows from your data source into a target search index. You should plan on creating one indexer for every target index and data source combination. While you can have multiple indexers writing into the same index, an indexer can only write into a single index. An indexer is used to:

- Perform a one-time copy of the data to populate an index.
- Sync an index with changes in the data source on a schedule. The schedule is part of the indexer definition.
- Invoke on-demand updates to an index as needed.

Step 1: Create a data source

Issue a HTTP POST request to create a new data source in your Azure Search service, including the following request headers.

```
POST https://[Search service name].search.windows.net/datasources?api-version=[api-version]
Content-Type: application/json
api-key: [Search service admin key]
```

The `api-version` is required. Valid values include `2015-02-28` or a later version. Visit [API versions in Azure Search](#) to see all supported Search API versions.

The body of the request contains the data source definition, which should include the following fields:

- **name:** Choose any name to represent your DocumentDB database.
- **type:** Use `documentdb`.
- **credentials:**

- **connectionString**: Required. Specify the connection info to your Azure DocumentDB database in the following format:

```
AccountEndpoint=<DocumentDB endpoint url>;AccountKey=<DocumentDB auth key>;Database=<DocumentDB database id>
```

- **container**:
 - **name**: Required. Specify the id of the DocumentDB collection to be indexed.
 - **query**: Optional. You can specify a query to flatten an arbitrary JSON document into a flat schema that Azure Search can index.
- **dataChangeDetectionPolicy**: Optional. See [Data Change Detection Policy](#) below.
- **dataDeletionDetectionPolicy**: Optional. See [Data Deletion Detection Policy](#) below.

See below for an [example request body](#).

Capturing changed documents

The purpose of a data change detection policy is to efficiently identify changed data items. Currently, the only supported policy is the **High Water Mark** policy using the **_ts** last-modified timestamp property provided by DocumentDB - which is specified as follows:

```
{
  "@odata.type" : "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
  "highWaterMarkColumnName" : "_ts"
}
```

You will also need to add **_ts** in the projection and **WHERE** clause for your query. For example:

```
SELECT s.id, s.Title, s.Abstract, s._ts FROM Sessions s WHERE s._ts >= @HighWaterMark
```

Capturing deleted documents

When rows are deleted from the source table, you should delete those rows from the search index as well. The purpose of a data deletion detection policy is to efficiently identify deleted data items. Currently, the only supported policy is the **Soft Delete** policy (deletion is marked with a flag of some sort), which is specified as follows:

```
{
  "@odata.type" : "#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
  "softDeleteColumnName" : "the property that specifies whether a document was deleted",
  "softDeleteMarkerValue" : "the value that identifies a document as deleted"
}
```

NOTE

You will need to include the `softDeleteColumnName` property in your `SELECT` clause if you are using a custom projection.

Request body example

The following example creates a data source with a custom query and policy hints:

```
{
  "name": "mydocdbdatasource",
  "type": "documentdb",
  "credentials": {
    "connectionString":
"AccountEndpoint=https://myDocDbEndpoint.documents.azure.com;AccountKey=myDocDbAuthKey;Database=myDocDbDatabaseId"
  },
  "container": {
    "name": "myDocDbCollectionId",
    "query": "SELECT s.id, s.Title, s.Abstract, s._ts FROM Sessions s WHERE s._ts > @HighWaterMark"
  },
  "dataChangeDetectionPolicy": {
    "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
    "highWaterMarkColumnName": "_ts"
  },
  "dataDeletionDetectionPolicy": {
    "@odata.type": "#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
    "softDeleteColumnName": "isDeleted",
    "softDeleteMarkerValue": "true"
  }
}
```

Response

You will receive an HTTP 201 Created response if the data source was successfully created.

Step 2: Create an index

Create a target Azure Search index if you don't have one already. You can do this from the [Azure Portal UI](#) or by using the [Create Index API](#).

```
POST https://[Search service name].search.windows.net/indexes?api-version=[api-version]
Content-Type: application/json
api-key: [Search service admin key]
```

Ensure that the schema of your target index is compatible with the schema of the source JSON documents or the output of your custom query projection.

NOTE

For partitioned collections, the default document key is DocumentDB's `_rid` property, which gets renamed to `rid` in Azure Search. Also, DocumentDB's `_rid` values contain characters that are invalid in Azure Search keys; therefore, the `_rid` values are Base64 encoded.

Figure A: Mapping between JSON Data Types and Azure Search Data Types

JSON DATA TYPE	COMPATIBLE TARGET INDEX FIELD TYPES
Bool	Edm.Boolean, Edm.String
Numbers that look like integers	Edm.Int32, Edm.Int64, Edm.String
Numbers that look like floating-points	Edm.Double, Edm.String
String	Edm.String
Arrays of primitive types e.g. "a", "b", "c"	Collection(Edm.String)

JSON DATA TYPE	COMPATIBLE TARGET INDEX FIELD TYPES
Strings that look like dates	Edm.DateTimeOffset, Edm.String
GeoJSON objects e.g. { "type": "Point", "coordinates": [long, lat] }	Edm.GeographyPoint
Other JSON objects	N/A

Request body example

The following example creates an index with an id and description field:

```
{
  "name": "mysearchindex",
  "fields": [{
    "name": "id",
    "type": "Edm.String",
    "key": true,
    "searchable": false
  }, {
    "name": "description",
    "type": "Edm.String",
    "filterable": false,
    "sortable": false,
    "facetable": false,
    "suggestions": true
  }]
}
```

Response

You will receive an HTTP 201 Created response if the index was successfully created.

Step 3: Create an indexer

You can create a new indexer within an Azure Search service by using an HTTP POST request with the following headers.

```
POST https://[Search service name].search.windows.net/indexers?api-version=[api-version]
Content-Type: application/json
api-key: [Search service admin key]
```

The body of the request contains the indexer definition, which should include the following fields:

- **name**: Required. The name of the indexer.
- **dataSourceName**: Required. The name of an existing data source.
- **targetIndexName**: Required. The name of an existing index.
- **schedule**: Optional. See [Indexing Schedule](#) below.

Running indexers on a schedule

An indexer can optionally specify a schedule. If a schedule is present, the indexer will run periodically as per schedule. Schedule has the following attributes:

- **interval**: Required. A duration value that specifies an interval or period for indexer runs. The smallest allowed interval is 5 minutes; the longest is one day. It must be formatted as an XSD "dayTimeDuration" value (a restricted subset of an [ISO 8601 duration](#) value). The pattern for this is: `P(nD)(T(nH)(nM))`. Examples: `PT15M` for every 15 minutes, `PT2H` for every 2 hours.

- **startTime**: Required. An UTC datetime that specifies when the indexer should start running.

Request body example

The following example creates an indexer that copies data from the collection referenced by the `myDocDbDataSource` data source to the `mySearchIndex` index on a schedule that starts on Jan 1, 2015 UTC and runs hourly.

```
{
  "name" : "mysearchindexer",
  "dataSourceName" : "mydocdbdatasource",
  "targetIndexName" : "mysearchindex",
  "schedule" : { "interval" : "PT1H", "startTime" : "2015-01-01T00:00:00Z" }
}
```

Response

You will receive an HTTP 201 Created response if the indexer was successfully created.

Step 4: Run an indexer

In addition to running periodically on a schedule, an indexer can also be invoked on demand by issuing the following HTTP POST request:

```
POST https://[Search service name].search.windows.net/indexers/[indexer name]/run?api-version=[api-version]
api-key: [Search service admin key]
```

Response

You will receive an HTTP 202 Accepted response if the indexer was successfully invoked.

Step 5: Get indexer status

You can issue a HTTP GET request to retrieve the current status and execution history of an indexer:

```
GET https://[Search service name].search.windows.net/indexers/[indexer name]/status?api-version=[api-version]
api-key: [Search service admin key]
```

Response

You will see a HTTP 200 OK response returned along with a response body that contains information about overall indexer health status, the last indexer invocation, as well as the history of recent indexer invocations (if present).

The response should look similar to the following:

```

{
  "status": "running",
  "lastResult": {
    "status": "success",
    "errorMessage": null,
    "startTime": "2014-11-26T03:37:18.853Z",
    "endTime": "2014-11-26T03:37:19.012Z",
    "errors": [],
    "itemsProcessed": 11,
    "itemsFailed": 0,
    "initialTrackingState": null,
    "finalTrackingState": null
  },
  "executionHistory": [ {
    "status": "success",
    "errorMessage": null,
    "startTime": "2014-11-26T03:37:18.853Z",
    "endTime": "2014-11-26T03:37:19.012Z",
    "errors": [],
    "itemsProcessed": 11,
    "itemsFailed": 0,
    "initialTrackingState": null,
    "finalTrackingState": null
  } ]
}

```

Execution history contains up to the 50 most recent completed executions, which are sorted in reverse chronological order (so the latest execution comes first in the response).

Next steps

Congratulations! You have just learned how to integrate Azure DocumentDB with Azure Search using the indexer for DocumentDB.

- To learn how more about Azure DocumentDB, see the [DocumentDB service page](#).
- To learn how more about Azure Search, see the [Search service page](#).

Move data to and from DocumentDB using Azure Data Factory

11/15/2016 • 9 min to read • [Edit on GitHub](#)

Contributors

Linda Wang • [Iain Foulds](#) • [James Dunn](#) • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [Sreedhar Pelluru](#)
• [Ross McAllister](#) • [mimig](#)

This article outlines how you can use the Copy Activity in an Azure data factory to move data to Azure DocumentDB from another data store and move data from DocumentDB to another data store. This article builds on the [data movement activities](#) article, which presents a general overview of data movement with copy activity and supported data store combinations.

The following samples show how to copy data to and from Azure DocumentDB and Azure Blob Storage. However, data can be copied **directly** from any of the sources to any of the supported sinks. For more information, see the section "Supported data stores and formats" in [Move data by using Copy Activity](#).

NOTE

Copying data from on-premises/Azure IaaS data stores to Azure DocumentDB and vice versa are supported with Data Management Gateway version 2.1 and above.

Supported versions

This DocumentDB connector support copying data from/to DocumentDB single partition collection and partitioned collection. [DocDB for MongoDB](#) is not supported.

Sample: Copy data from DocumentDB to Azure Blob

The sample below shows:

1. A linked service of type [DocumentDb](#).
2. A linked service of type [AzureStorage](#).
3. An input [dataset](#) of type [DocumentDbCollection](#).
4. An output [dataset](#) of type [AzureBlob](#).
5. A [pipeline](#) with Copy Activity that uses [DocumentDbCollectionSource](#) and [BlobSink](#).

The sample copies data in Azure DocumentDB to Azure Blob. The JSON properties used in these samples are described in sections following the samples.

Azure DocumentDB linked service:

```
{
  "name": "DocumentDbLinkedService",
  "properties": {
    "type": "DocumentDb",
    "typeProperties": {
      "connectionString": "AccountEndpoint=<EndpointUrl>;AccountKey=<AccessKey>;Database=<Database>"
    }
  }
}
```

Azure Blob storage linked service:

```
{
  "name": "StorageLinkedService",
  "properties": {
    "type": "AzureStorage",
    "typeProperties": {
      "connectionString": "DefaultEndpointsProtocol=https;AccountName=<accountname>;AccountKey=<accountkey>"
    }
  }
}
```

Azure Document DB input dataset:

The sample assumes you have a collection named **Person** in an Azure DocumentDB database.

Setting "external": "true" and specifying externalData policy information the Azure Data Factory service that the table is external to the data factory and not produced by an activity in the data factory.

```
{
  "name": "PersonDocumentDbTable",
  "properties": {
    "type": "DocumentDbCollection",
    "linkedServiceName": "DocumentDbLinkedService",
    "typeProperties": {
      "collectionName": "Person"
    },
    "external": true,
    "availability": {
      "frequency": "Day",
      "interval": 1
    }
  }
}
```

Azure Blob output dataset:

Data is copied to a new blob every hour with the path for the blob reflecting the specific datetime with hour granularity.

```
{
  "name": "PersonBlobTableOut",
  "properties": {
    "type": "AzureBlob",
    "linkedServiceName": "StorageLinkedService",
    "typeProperties": {
      "folderPath": "docdb",
      "format": {
        "type": "TextFormat",
        "columnDelimiter": ",",
        "nullValue": "NULL"
      }
    },
    "availability": {
      "frequency": "Day",
      "interval": 1
    }
  }
}
```

Sample JSON document in the Person collection in a DocumentDB database:

```
{
  "PersonId": 2,
  "Name": {
    "First": "Jane",
    "Middle": "",
    "Last": "Doe"
  }
}
```

DocumentDB supports querying documents using a SQL like syntax over hierarchical JSON documents.

Example:

```
SELECT Person.PersonId, Person.Name.First AS FirstName, Person.Name.Middle as MiddleName, Person.Name.Last AS
LastName FROM Person
```

The following pipeline copies data from the Person collection in the DocumentDB database to an Azure blob. As part of the copy activity the input and output datasets have been specified.

```

{
  "name": "DocDbToBlobPipeline",
  "properties": {
    "activities": [
      {
        "type": "Copy",
        "typeProperties": {
          "source": {
            "type": "DocumentDbCollectionSource",
            "query": "SELECT Person.Id, Person.Name.First AS FirstName, Person.Name.Middle as MiddleName,
Person.Name.Last AS LastName FROM Person",
            "nestingSeparator": "."
          },
          "sink": {
            "type": "BlobSink",
            "blobWriterAddHeader": true,
            "writeBatchSize": 1000,
            "writeBatchTimeout": "00:00:59"
          }
        },
        "inputs": [
          {
            "name": "PersonDocumentDbTable"
          }
        ],
        "outputs": [
          {
            "name": "PersonBlobTableOut"
          }
        ],
        "policy": {
          "concurrency": 1
        },
        "name": "CopyFromDocDbToBlob"
      }
    ],
    "start": "2015-04-01T00:00:00Z",
    "end": "2015-04-02T00:00:00Z"
  }
}

```

Sample: Copy data from Azure Blob to Azure DocumentDB

The sample below shows:

1. A linked service of type [DocumentDb](#).
2. A linked service of type [AzureStorage](#).
3. An input [dataset](#) of type [AzureBlob](#).
4. An output [dataset](#) of type [DocumentDbCollection](#).
5. A [pipeline](#) with Copy Activity that uses [BlobSource](#) and [DocumentDbCollectionSink](#).

The sample copies data from Azure blob to Azure DocumentDB. The JSON properties used in these samples are described in sections following the samples.

Azure Blob storage linked service:

```
{
  "name": "StorageLinkedService",
  "properties": {
    "type": "AzureStorage",
    "typeProperties": {
      "connectionString": "DefaultEndpointsProtocol=https;AccountName=<accountname>;AccountKey=<accountkey>"
    }
  }
}
```

Azure DocumentDB linked service:

```
{
  "name": "DocumentDbLinkedService",
  "properties": {
    "type": "DocumentDb",
    "typeProperties": {
      "connectionString": "AccountEndpoint=<EndpointUrl>;AccountKey=<AccessKey>;Database=<Database>"
    }
  }
}
```

Azure Blob input dataset:

```
{
  "name": "PersonBlobTableIn",
  "properties": {
    "structure": [
      {
        "name": "Id",
        "type": "Int"
      },
      {
        "name": "FirstName",
        "type": "String"
      },
      {
        "name": "MiddleName",
        "type": "String"
      },
      {
        "name": "LastName",
        "type": "String"
      }
    ],
    "type": "AzureBlob",
    "linkedServiceName": "StorageLinkedService",
    "typeProperties": {
      "fileName": "input.csv",
      "folderPath": "docdb",
      "format": {
        "type": "TextFormat",
        "columnDelimiter": ";",
        "nullValue": "NULL"
      }
    },
    "external": true,
    "availability": {
      "frequency": "Day",
      "interval": 1
    }
  }
}
```

Azure DocumentDB output dataset:

The sample copies data to a collection named "Person".

```
{
  "name": "PersonDocumentDbTableOut",
  "properties": {
    "structure": [
      {
        "name": "Id",
        "type": "Int"
      },
      {
        "name": "Name.First",
        "type": "String"
      },
      {
        "name": "Name.Middle",
        "type": "String"
      },
      {
        "name": "Name.Last",
        "type": "String"
      }
    ],
    "type": "DocumentDbCollection",
    "linkedServiceName": "DocumentDbLinkedService",
    "typeProperties": {
      "collectionName": "Person"
    },
    "availability": {
      "frequency": "Day",
      "interval": 1
    }
  }
}
```

The following pipeline copies data from Azure Blob to the Person collection in the DocumentDB. As part of the copy activity the input and output datasets have been specified.

```

{
  "name": "BlobToDocDbPipeline",
  "properties": {
    "activities": [
      {
        "type": "Copy",
        "typeProperties": {
          "source": {
            "type": "BlobSource"
          },
          "sink": {
            "type": "DocumentDbCollectionSink",
            "nestingSeparator": ".",
            "writeBatchSize": 2,
            "writeBatchTimeout": "00:00:00"
          }
        },
        "translator": {
          "type": "TabularTranslator",
          "ColumnMappings": "FirstName: Name.First, MiddleName: Name.Middle, LastName: Name.Last,
BusinessEntityID: BusinessEntityID, PersonType: PersonType, NameStyle: NameStyle, Title: Title, Suffix: Suffix,
EmailPromotion: EmailPromotion, rowguid: rowguid, ModifiedDate: ModifiedDate"
        }
      },
      {
        "inputs": [
          {
            "name": "PersonBlobTableIn"
          }
        ],
        "outputs": [
          {
            "name": "PersonDocumentDbTableOut"
          }
        ],
        "policy": {
          "concurrency": 1
        },
        "name": "CopyFromBlobToDocDb"
      }
    ],
    "start": "2015-04-14T00:00:00Z",
    "end": "2015-04-15T00:00:00Z"
  }
}

```

If the sample blob input is as

```
1,John,,Doe
```

Then the output JSON in DocumentDB will be as:

```

{
  "Id": 1,
  "Name": {
    "First": "John",
    "Middle": null,
    "Last": "Doe"
  },
  "id": "a5e8595c-62ec-4554-a118-3940f4ff70b6"
}

```

DocumentDB is a NoSQL store for JSON documents, where nested structures are allowed. Azure Data Factory enables user to denote hierarchy via **nestingSeparator**, which is "." in this example. With the separator, the copy activity will generate the "Name" object with three children elements First, Middle and Last, according to "Name.First", "Name.Middle" and "Name.Last" in the table definition.

Azure DocumentDB Linked Service properties

The following table provides description for JSON elements specific to Azure DocumentDB linked service.

PROPERTY	DESCRIPTION	REQUIRED
type	The type property must be set to: DocumentDb	Yes
connectionString	Specify information needed to connect to Azure DocumentDB database.	Yes

Azure DocumentDB Dataset type properties

For a full list of sections & properties available for defining datasets please refer to the [Creating datasets](#) article. Sections like structure, availability, and policy of a dataset JSON are similar for all dataset types (Azure SQL, Azure blob, Azure table, etc.).

The typeProperties section is different for each type of dataset and provides information about the location of the data in the data store. The typeProperties section for the dataset of type **DocumentDbCollection** has the following properties.

PROPERTY	DESCRIPTION	REQUIRED
collectionName	Name of the DocumentDB document collection.	Yes

Example:

```
{
  "name": "PersonDocumentDbTable",
  "properties": {
    "type": "DocumentDbCollection",
    "linkedServiceName": "DocumentDbLinkedService",
    "typeProperties": {
      "collectionName": "Person"
    },
    "external": true,
    "availability": {
      "frequency": "Day",
      "interval": 1
    }
  }
}
```

Schema by Data Factory

For schema-free data stores such as DocumentDB, the Data Factory service infers the schema in one of the following ways:

1. If you specify the structure of data by using the **structure** property in the dataset definition, the Data Factory service honors this structure as the schema. In this case, if a row does not contain a value for a column, a null value will be provided for it.
2. If you do not specify the structure of data by using the **structure** property in the dataset definition, the Data Factory service infers the schema by using the first row in the data. In this case, if the first row does not contain the full schema, some columns will be missing in the result of copy operation.

Therefore, for schema-free data sources, the best practice is to specify the structure of data using the **structure** property.

Azure DocumentDB Copy Activity type properties

For a full list of sections & properties available for defining activities please refer to the [Creating Pipelines](#) article. Properties such as name, description, input and output tables, and policy are available for all types of activities.

Note: The Copy Activity takes only one input and produces only one output.

Properties available in the typeProperties section of the activity on the other hand vary with each activity type and in case of Copy activity they vary depending on the types of sources and sinks.

In case of Copy activity when source is of type **DocumentDbCollectionSource** the following properties are available in **typeProperties** section:

PROPERTY	DESCRIPTION	ALLOWED VALUES	REQUIRED
query	Specify the query to read data.	Query string supported by DocumentDB. Example: <pre>SELECT c.BusinessEntityID, c.PersonType, c.NameStyle, c.Title, c.Name.First AS FirstName, c.Name.Last AS LastName, c.Suffix, c.EmailPromotion FROM c WHERE c.ModifiedDate > \"2009-01-01T00:00:00\"</pre>	No If not specified, the SQL statement that is executed: <pre>select <columns defined in structure> from mycollection</pre>
nestingSeparator	Special character to indicate that the document is nested	Any character. DocumentDB is a NoSQL store for JSON documents, where nested structures are allowed. Azure Data Factory enables user to denote hierarchy via nestingSeparator, which is "." in the above examples. With the separator, the copy activity will generate the "Name" object with three children elements First, Middle and Last, according to "Name.First", "Name.Middle" and "Name.Last" in the table definition.	No

DocumentDbCollectionSink supports the following properties:

PROPERTY	DESCRIPTION	ALLOWED VALUES	REQUIRED
----------	-------------	----------------	----------

PROPERTY	DESCRIPTION	ALLOWED VALUES	REQUIRED
nestingSeparator	<p>A special character in the source column name to indicate that nested document is needed.</p> <p>For example above: <code>Name.First</code> in the output table produces the following JSON structure in the DocumentDB document:</p> <pre>"Name": { "First": "John" },</pre>	<p>Character that is used to separate nesting levels.</p> <p>Default value is <code>.</code> (dot).</p>	<p>Character that is used to separate nesting levels.</p> <p>Default value is <code>.</code> (dot).</p>
writeBatchSize	<p>Number of parallel requests to DocumentDB service to create documents.</p> <p>You can fine-tune the performance when copying data to/from DocumentDB by using this property. You can expect a better performance when you increase writeBatchSize because more parallel requests to DocumentDB are sent. However you'll need to avoid throttling that can throw the error message: "Request rate is large".</p> <p>Throttling is decided by a number of factors, including size of documents, number of terms in documents, indexing policy of target collection, etc. For copy operations, you can use a better collection (e.g. S3) to have the most throughput available (2,500 request units/second).</p>	Integer	No (default: 5)
writeBatchTimeout	Wait time for the operation to complete before it times out.	<p>timespan</p> <p>Example: "00:30:00" (30 minutes).</p>	No

Import/Export JSON documents

Using this DocumentDB connector, you can easily

- Import JSON documents from various sources into DocumentDB, including Azure Blob, Azure Data Lake, on-prem File System or other file-based stores supported by Azure Data Factory
- Export JSON documents from DocumentDB collection into various file-based stores
- Migrate data between two DocumentDB collections as-is

To achieve such schema-agnostic copy, do not specify the "structure" section in input dataset or "nestingSeparator" property on DocumentDB source/sink in copy activity. See "Specify format" section in corresponding file-based connector topic on JSON format configuration details.

Appendix

1. **Question:** Does the Copy Activity support update of existing records?

Answer: No.

2. **Question:** How does a retry of a copy to DocumentDB deal with already copied records?

Answer: If records have an "ID" field and the copy operation tries to insert a record with the same ID, the copy operation throws an error.

3. **Question:** Does Data Factory support [range or hash-based data partitioning](#)?

Answer: No.

4. **Question:** Can I specify more than one DocumentDB collection for a table?

Answer: No. Only one collection can be specified at this time.

Performance and Tuning

See [Copy Activity Performance & Tuning Guide](#) to learn about key factors that impact performance of data movement (Copy Activity) in Azure Data Factory and various ways to optimize it.

Stream Analytics outputs: Options for storage, analysis

11/15/2016 • 14 min to read • [Edit on GitHub](#)

Contributors

[Jeff Stokes](#) • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [Larry Franks](#) • [Carolyn Gronlund](#) • [v-aljenk](#)

When authoring a Stream Analytics job, consider how the resulting data will be consumed. How will you view the results of the Stream Analytics job and where will you store it?

In order to enable a variety of application patterns, Azure Stream Analytics has different options for storing output and viewing analysis results. This makes it easy to view job output and gives you flexibility in the consumption and storage of the job output for data warehousing and other purposes. Any output configured in the job must exist before the job is started and events start flowing. For example, if you use Blob storage as an output, the job will not create a storage account automatically. It needs to be created by the user before the ASA job is started.

Azure Data Lake Store

Stream Analytics supports [Azure Data Lake Store](#). This storage enables you to store data of any size, type and ingestion speed for operational and exploratory analytics. At this time, creation and configuration of Data Lake Store outputs is supported only in the Azure Classic Portal. Further, Stream Analytics needs to be authorized to access the Data Lake Store. Details on authorization and how to sign up for the Data Lake Store Preview (if needed) are discussed in the [Data Lake output article](#).

Authorize an Azure Data Lake Store

When Data Lake Storage is selected as an output in the Azure Management portal, you will be prompted to authorize a connection to an existing Data Lake Store.

New output

* Output alias

datalakeoutput

✓

* Sink ⓘ

Data Lake Store

▼

Authorize Connection

You'll need to authorize with Data Lake Store to configure your output settings.

Authorize

Don't have a Microsoft Azure Data Lake Store account yet?
[Sign Up](#)

i


Note: You are granting this output permanent access to your Data Lake Store account. Should you need to revoke this access in the future you can do one of the following:

1. Change the user account password.
2. Delete this output.
3. Delete this job.

Create

Then fill out the properties for the Data Lake Store output as seen below:

New output



The selected resource and the stream analytics job are located in different regions. You will be billed to move data between regions.

* Output alias

adlsoutput

✓

* Sink ⓘ

Data Lake Store

▼

* Account Name

adlajsdemo

▼

* Path prefix pattern

Date format

YYYY/MM/DD

▼

Time format

HH

▼

* Event serialization format ⓘ

JSON

▼

Encoding ⓘ

UTF-8

▼

Format ⓘ

Line separated

▼

Create

The table below lists the property names and their description needed for creating a Data Lake Store output.

PROPERTY NAME	DESCRIPTION
Output Alias	This is a friendly name used in queries to direct the query output to this Data Lake Store.
Account Name	The name of the Data Lake Storage account where you are sending your output. You will be presented with a drop down list of Data Lake Store accounts to which the user logged in to the portal has access to.

Path Prefix Pattern <i>[optional]</i>	<p>The file path used to write your files within the specified Data Lake Store Account.</p> <p>{date}, {time}</p> <p>Example 1: folder1/logs/{date}/{time}</p> <p>Example 2: folder1/logs/{date}</p>
Date Format <i>[optional]</i>	<p>If the date token is used in the prefix path, you can select the date format in which your files are organized. Example: YYYY/MM/DD</p>
Time Format <i>[optional]</i>	<p>If the time token is used in the prefix path, specify the time format in which your files are organized. Currently the only supported value is HH.</p>
Event Serialization Format	<p>Serialization format for output data. JSON, CSV, and Avro are supported.</p>
Encoding	<p>If CSV or JSON format, an encoding must be specified. UTF-8 is the only supported encoding format at this time.</p>
Delimiter	<p>Only applicable for CSV serialization. Stream Analytics supports a number of common delimiters for serializing CSV data. Supported values are comma, semicolon, space, tab and vertical bar.</p>
Format	<p>Only applicable for JSON serialization. Line separated specifies that the output will be formatted by having each JSON object separated by a new line. Array specifies that the output will be formatted as an array of JSON objects.</p>

Renew Data Lake Store Authorization

You will need to re-authenticate your Data Lake Store account if its password has changed since your job was created or last authenticated.

Output details

adlsoutput

Test

Delete

★ Path prefix pattern

/asd/asd

Date format

YYYY/MM/DD

Time format

HH

★ Event serialization format ⓘ

JSON

Encoding ⓘ

UTF-8

Format ⓘ

Line separated

Authorization

Click the button below if you want to renew authorization or authorize with a different account.

Renew authorization

i

Note: This output has permanent access to your Data Lake Store account. Access to Data Lake, once granted, does not expire unless you do one of the following:

1. Change the user account password.

2. Delete this output.

3. Delete this job.

Save

SQL Database

[Azure SQL Database](#) can be used as an output for data that is relational in nature or for applications that depend on content being hosted in a relational database. Stream Analytics jobs will write to an existing table in an Azure SQL Database. Note that the table schema must exactly match the fields and their types being output from your job. An [Azure SQL Data Warehouse](#) can also be specified as an output via the SQL Database output option as well (this is a preview feature). The table below lists the property names and their description for creating a SQL Database output.

PROPERTY NAME	DESCRIPTION
Output Alias	This is a friendly name used in queries to direct the query output to this database.
Database	The name of the database where you are sending your output

PROPERTY NAME	DESCRIPTION
Server Name	The SQL Database server name
Username	The Username which has access to write to the database
Password	The password to connect to the database
Table	The table name where the output will be written. The table name is case sensitive and the schema of this table should match exactly to the number of fields and their types being generated by your job output.

NOTE

Currently the Azure SQL Database offering is supported for a job output in Stream Analytics. However, an Azure Virtual Machine running SQL Server with a database attached is not supported. This is subject to change in future releases.

Blob storage

Blob storage offers a cost-effective and scalable solution for storing large amounts of unstructured data in the cloud. For an introduction on Azure Blob storage and its usage, see the documentation at [How to use Blobs](#).

The table below lists the property names and their description for creating a blob output.

PROPERTY NAME	DESCRIPTION
Output Alias	This is a friendly name used in queries to direct the query output to this blob storage.
Storage Account	The name of the storage account where you are sending your output.
Storage Account Key	The secret key associated with the storage account.
Storage Container	Containers provide a logical grouping for blobs stored in the Microsoft Azure Blob service. When you upload a blob to the Blob service, you must specify a container for that blob.
Path Prefix Pattern [optional]	<p>The file path used to write your blobs within the specified container.</p> <p>Within the path, you may choose to use one or more instances of the following 2 variables to specify the frequency that blobs are written:</p> <p>{date}, {time}</p> <p>Example 1: cluster1/logs/{date}/{time}</p> <p>Example 2: cluster1/logs/{date}</p>
Date Format [optional]	If the date token is used in the prefix path, you can select the date format in which your files are organized. Example: YYYY/MM/DD
Time Format [optional]	If the time token is used in the prefix path, specify the time format in which your files are organized. Currently the only supported value is HH.

Event Serialization Format	Serialization format for output data. JSON, CSV, and Avro are supported.
Encoding	If CSV or JSON format, an encoding must be specified. UTF-8 is the only supported encoding format at this time.
Delimiter	Only applicable for CSV serialization. Stream Analytics supports a number of common delimiters for serializing CSV data. Supported values are comma, semicolon, space, tab and vertical bar.
Format	Only applicable for JSON serialization. Line separated specifies that the output will be formatted by having each JSON object separated by a new line. Array specifies that the output will be formatted as an array of JSON objects.

Event Hub

[Event Hubs](#) is a highly scalable publish-subscribe event ingestor. It can collect millions of events per second. One use of an Event Hub as output is when the output of a Stream Analytics job will be the input of another streaming job.

There are a few parameters that are needed to configure Event Hub data streams as an output.

PROPERTY NAME	DESCRIPTION
Output Alias	This is a friendly name used in queries to direct the query output to this Event Hub.
Service Bus Namespace	A Service Bus namespace is a container for a set of messaging entities. When you created a new Event Hub, you also created a Service Bus namespace
Event Hub	The name of your Event Hub output
Event Hub Policy Name	The shared access policy, which can be created on the Event Hub Configure tab. Each shared access policy will have a name, permissions that you set, and access keys
Event Hub Policy Key	The Shared Access key used to authenticate access to the Service Bus namespace
Partition Key Column [optional]	This column contains the partition key for Event Hub output.
Event Serialization Format	Serialization format for output data. JSON, CSV, and Avro are supported.
Encoding	For CSV and JSON, UTF-8 is the only supported encoding format at this time
Delimiter	Only applicable for CSV serialization. Stream Analytics supports a number of common delimiters for serializing data in CSV format. Supported values are comma, semicolon, space, tab and vertical bar.

PROPERTY NAME	DESCRIPTION
Format	Only applicable for JSON type. Line separated specifies that the output will be formatted by having each JSON object separated by a new line. Array specifies that the output will be formatted as an array of JSON objects.

Power BI

Power BI can be used as an output for a Stream Analytics job to provide for a rich visualization experience of analysis results. This capability can be used for operational dashboards, report generation and metric driven reporting.

Authorize a Power BI account

1. When Power BI is selected as an output in the Azure Management portal, you will be prompted to authorize an existing Power BI User or to create a new Power BI account.

New output

* Output alias
pbioutput ✓

* Sink ⓘ
Power BI ▼

Authorize Connection

You'll need to authorize with Power BI to configure your output settings.

[Authorize](#)

Don't have a Microsoft Power BI account yet?
[Sign Up](#)

Note: You are granting this output permanent access to your Power BI dashboard. Should you need to revoke this access in the future you can do one of the following:

1. Change the user account password.
2. Delete this output.
3. Delete this job.

[Create](#)

2. Create a new account if you don't yet have one, then click Authorize Now. A screen like the following is presented.

3. In this step, provide the work or school account for authorizing the Power BI output. If you are not already signed up for Power BI, choose Sign up now. The work or school account you use for Power BI could be different from the Azure subscription account which you are currently logged in with.

Configure the Power BI output properties

Once you have the Power BI account authenticated, you can configure the properties for your Power BI output. The table below is the list of property names and their description to configure your Power BI output.

PROPERTY NAME	DESCRIPTION
Output Alias	This is a friendly name used in queries to direct the query output to this PowerBI output.
Group Workspace	To enable sharing data with other Power BI users you can select groups inside your Power BI account or choose "My Workspace" if you do not want to write to a group. Updating an existing group requires renewing the Power BI authentication.
Dataset Name	Provide a dataset name that it is desired for the Power BI output to use
Table Name	Provide a table name under the dataset of the Power BI output. Currently, Power BI output from Stream Analytics jobs can only have one table in a dataset

For a walk-through of configuring a Power BI output and dashboard, please see the [Azure Stream Analytics & Power BI](#) article.

NOTE

Do not explicitly create the dataset and table in the Power BI dashboard. The dataset and table will be automatically populated when the job is started and the job starts pumping output into Power BI. Note that if the job query doesn't generate any results, the dataset and table will not be created. Also be aware that if Power BI already had a dataset and table with the same name as the one provided in this Stream Analytics job, the existing data will be overwritten.

Renew Power BI Authorization

You will need to re-authenticate your Power BI account if its password has changed since your job was created or

last authenticated. If Multi-Factor Authentication (MFA) is configured on your Azure Active Directory (AAD) tenant you will also need to renew Power BI authorization every 2 weeks. A symptom of this issue is no job output and an "Authenticate user error, please re-" in the Operation Logs:

5/10/2015 7:34:27 PM	Send Events	Failed	BluetoothSensorToPBI	Microsoft.StreamAnaly...	81c115ac-d3e1-42f8-8d...
5/10/2015 7:34:27 PM	Initialize Adapter	Failed	BluetoothSensorToPBI	Microsoft.StreamAnaly...	69170270-4b08-45a6-b7...
5/10/2015 7:34:27 PM	Authenticate user error, please re-	Failed	BluetoothSensorToPBI	Microsoft.StreamAnaly...	27b68caf-ccd8-4b22-8e...
5/10/2015 7:34:26 PM	Send Events	Failed	BluetoothSensorToPBI	Microsoft.StreamAnaly...	0d3bf0ef-2c99-492d-b4...
5/10/2015 7:34:26 PM	Initialize Adapter	Failed	BluetoothSensorToPBI	Microsoft.StreamAnaly...	095f0cc6-ed74-45e7-b6...
5/10/2015 7:34:26 PM	Initialize Adapter	Failed	BluetoothSensorToPBI	Microsoft.StreamAnaly...	6417a953-3753-4960-b9...

To resolve this issue, stop your running job and go to your Power BI output. Click the "Renew authorization" link, and restart your job from the Last Stopped Time to avoid data loss.

Output details

pbiooutput

Test

Delete

Group Workspace

My Workspace

* Dataset Name

datasetname

If the dataset or table already exists in yo...
Microsoft Power BI subscription, it will be
overwritten.

* Table Name

tablename

Authorization

Click the button below if you want to renew
authorization, authorize with a different account
or modify the workspace.

Renew authorization

i

Note: This output has permanent
access to your Power BI dashboard.
Access to Power BI, once granted,
does not expire unless you do one
of the following:

1. Change the user account
password.

2. Delete this output.

3. Delete this job.

Save

Table Storage

Azure Table storage offers highly available, massively scalable storage, so that an application can automatically

scale to meet user demand. Table storage is Microsoft's NoSQL key/attribute store which one can leverage for structured data with less constraints on the schema. Azure Table storage can be used to store data for persistence and efficient retrieval.

The table below lists the property names and their description for creating a table output.

PROPERTY NAME	DESCRIPTION
Output Alias	This is a friendly name used in queries to direct the query output to this table storage.
Storage Account	The name of the storage account where you are sending your output.
Storage Account Key	The access key associated with the storage account.
Table Name	The name of the table. The table will get created if it does not exist.
Partition Key	The name of the output column containing the partition key. The partition key is a unique identifier for the partition within a given table that forms the first part of an entity's primary key. It is a string value that may be up to 1 KB in size.
Row Key	The name of the output column containing the row key. The row key is a unique identifier for an entity within a given partition. It forms the second part of an entity's primary key. The row key is a string value that may be up to 1 KB in size.
Batch Size	The number of records for a batch operation. Typically the default is sufficient for most jobs, refer to the Table Batch Operation spec for more details on modifying this setting.

Service Bus Queues

[Service Bus Queues](#) offer a First In, First Out (FIFO) message delivery to one or more competing consumers.

Typically, messages are expected to be received and processed by the receivers in the temporal order in which they were added to the queue, and each message is received and processed by only one message consumer.

The table below lists the property names and their description for creating a Queue output.

PROPERTY NAME	DESCRIPTION
Output Alias	This is a friendly name used in queries to direct the query output to this Service Bus Queue.
Service Bus Namespace	A Service Bus namespace is a container for a set of messaging entities.
Queue Name	The name of the Service Bus Queue.
Queue Policy Name	When you create a Queue, you can also create shared access policies on the Queue Configure tab. Each shared access policy will have a name, permissions that you set, and access keys.

PROPERTY NAME	DESCRIPTION
Queue Policy Key	The Shared Access key used to authenticate access to the Service Bus namespace
Event Serialization Format	Serialization format for output data. JSON, CSV, and Avro are supported.
Encoding	For CSV and JSON, UTF-8 is the only supported encoding format at this time
Delimiter	Only applicable for CSV serialization. Stream Analytics supports a number of common delimiters for serializing data in CSV format. Supported values are comma, semicolon, space, tab and vertical bar.
Format	Only applicable for JSON type. Line separated specifies that the output will be formatted by having each JSON object separated by a new line. Array specifies that the output will be formatted as an array of JSON objects.

Service Bus Topics

While Service Bus Queues provide a one to one communication method from sender to receiver, [Service Bus Topics](#) provide a one-to-many form of communication.

The table below lists the property names and their description for creating a table output.

PROPERTY NAME	DESCRIPTION
Output Alias	This is a friendly name used in queries to direct the query output to this Service Bus Topic.
Service Bus Namespace	A Service Bus namespace is a container for a set of messaging entities. When you created a new Event Hub, you also created a Service Bus namespace
Topic Name	Topics are messaging entities, similar to event hubs and queues. They're designed to collect event streams from a number of different devices and services. When a topic is created, it is also given a specific name. The messages sent to a Topic will not be available unless a subscription is created, so ensure there are one or more subscriptions under the topic
Topic Policy Name	When you create a Topic, you can also create shared access policies on the Topic Configure tab. Each shared access policy will have a name, permissions that you set, and access keys
Topic Policy Key	The Shared Access key used to authenticate access to the Service Bus namespace
Event Serialization Format	Serialization format for output data. JSON, CSV, and Avro are supported.
Encoding	If CSV or JSON format, an encoding must be specified. UTF-8 is the only supported encoding format at this time

PROPERTY NAME	DESCRIPTION
Delimiter	Only applicable for CSV serialization. Stream Analytics supports a number of common delimiters for serializing data in CSV format. Supported values are comma, semicolon, space, tab and vertical bar.

DocumentDB

[Azure DocumentDB](#) is a fully-managed NoSQL document database service that offers query and transactions over schema-free data, predictable and reliable performance, and rapid development.

The table below lists the property names and their description for creating a DocumentDB output.

PROPERTY NAME	DESCRIPTION
Account Name	The name of the DocumentDB account. This can also be the endpoint for the account.
Account Key	The shared access key for the DocumentDB account.
Database	The DocumentDB database name.
Collection Name Pattern	The collection name pattern for the collections to be used. The collection name format can be constructed using the optional {partition} token, where partitions start from 0. E.g. The followings are valid inputs: MyCollection{partition} MyCollection Note that collections must exist before the Stream Analytics job is started and will not be created automatically.
Partition Key	The name of the field in output events used to specify the key for partitioning output across collections.
Document ID	The name of the field in output events used to specify the primary key which insert or update operations are based on.

Get help

For further assistance, try our [Azure Stream Analytics forum](#)

Next steps

You've been introduced to Stream Analytics, a managed service for streaming analytics on data from the Internet of Things. To learn more about this service, see:

- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Notifications for new or changed DocumentDB resources using Logic Apps

11/15/2016 • 16 min to read • [Edit on GitHub](#)

Contributors

[Howard S. Edidin](#) • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [mimig](#)

This article came about from a question I saw posted one of the Azure DocumentDB community forums. The question was **Does DocumentDB support notifications for modified resources?**

I have worked with BizTalk Server for many years, and this is a very common scenario when using the [WCF LOB Adapter](#). So I decided to see if I could duplicate this functionality in DocumentDB for new and/or modified documents.

This article provides an overview of the components of the change notification solution, which includes a [trigger](#) and a [Logic App](#). Important code snippets are provided inline and the entire solution is available on [GitHub](#).

Use case

The following story is the use case for this article.

DocumentDB is the repository for Health Level Seven International (HL7) Fast Healthcare Interoperability Resources (FHIR) documents. Let's assume that your DocumentDB database combined with your API and Logic App make up an HL7 FHIR Server. A healthcare facility is storing patient data in the DocumentDB "Patients" database. There are several collections within the patient database; Clinical, Identification, etc. Patient information falls under identification. You have a collection named "Patient".

The Cardiology department is tracking personal health and exercise data. Searching for new or modified Patient records is time consuming. They asked the IT department if there was a way that they could receive a notification for new or modified Patient records.

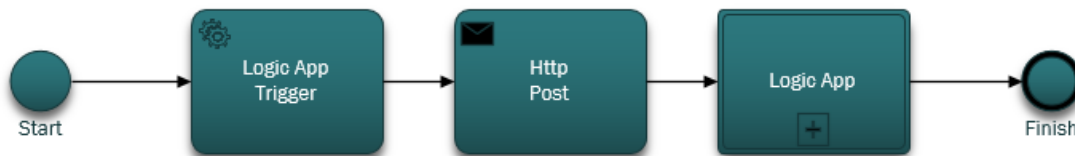
The IT department said that they could easily provide this. They also said that they could push the documents to [Azure Blob Storage](#) so the Cardiology department could easily access them.

How the IT department solved the problem

In order to create this application, the IT department decided to model it first. The nice thing about using Business Process Model and Notation (BPMN) is that both technical and non-technical people can easily understand it. This whole notification process is considered a business process.

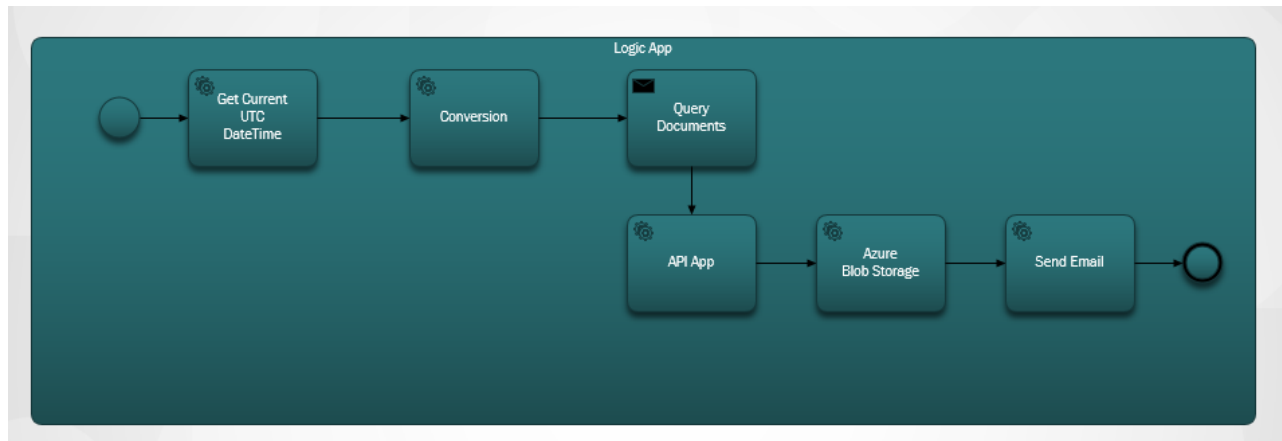
High-Level view of notification process

1. You start with a Logic App that has a timer trigger. By default, the trigger runs every hour.
2. Next you do an HTTP POST to the Logic App.
3. The Logic App does all the work.



Let's take a look at what this Logic App does

If you look at the following figure there are several steps in the LogicApp workflow.



The steps are as follows:

1. You need to get the current UTC DateTime from an API App. The default value is one hour previous.
2. The UTC DateTime is converted to a Unix Timestamp format. This is the default format for timestamps in DocumentDB.
3. You POST the value to an API App, which does a DocumentDB query. The value is used in a query.

```
SELECT * FROM Patients p WHERE (p._ts >= @unixTimeStamp)
```

NOTE

The `_ts` represents the TimeStamp metadata for all DocumentDB resources.

4. If there are documents found, the response body is sent to your Azure Blob Storage.

NOTE

Blob storage requires an Azure Storage account. You need to provision an Azure Blob storage account and add a new Blob named patients. For more information, see [About Azure storage accounts](#) and [Get started with Azure Blob storage](#).

5. Finally, an email is sent that notifies the recipient of the number of documents found. If no documents were found, the email body would be "0 Documents Found".

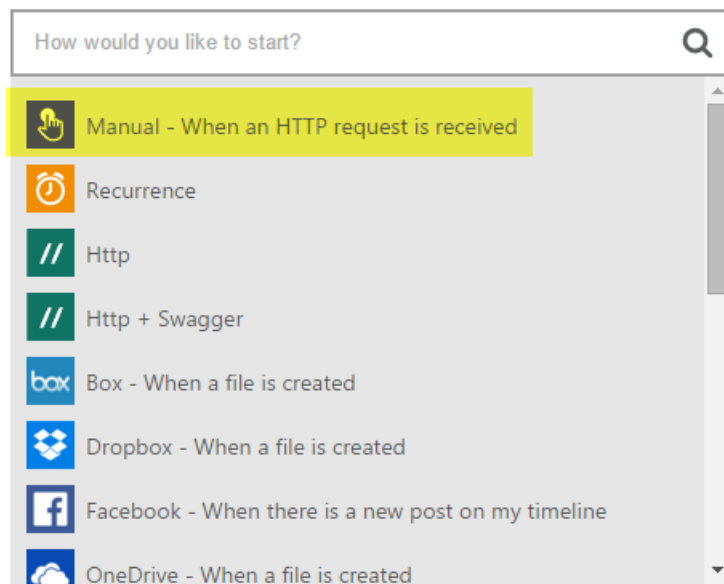
Now that you have an idea of what the workflow does, let's take a look at how you implement it.

Let's start with the main Logic App

If you're not familiar with Logic Apps, they are available in the [Azure Marketplace](#), and you can learn more about them in [What are Logic Apps?](#)

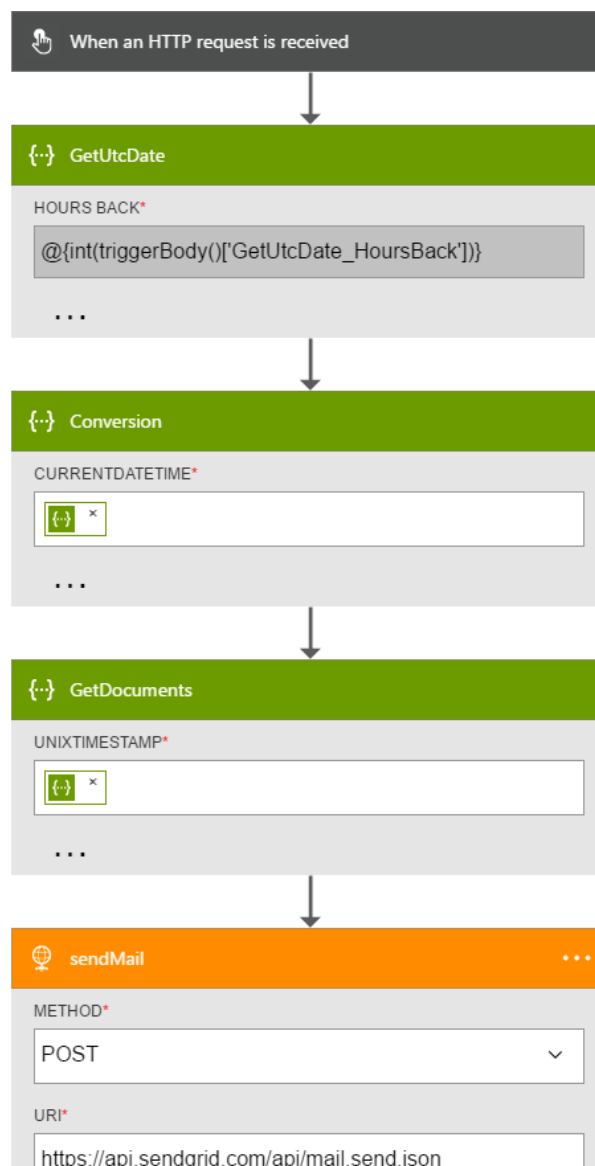
When you create a new Logic App, you are asked **How would you like to start?**

When you click inside the text box, you have a choice of events. For this Logic App, select **Manual - When an HTTP request is received** as shown below.



Design View of your completed Logic App

Let's jump ahead and look at the completed design view for the Logic App, which is named DocDB.



HEADERS

{"Content-type":"application/x-www-form-urlencoded"}

BODY

api_user=@{triggerBody()['_sendgridUsername']}&api_key=@{triggerBody()['_sendgridPassword']}&from=@{parameters('fromAddress')}&to=@{triggerBody()['_EmailTo']}&subject=@{triggerBody()['_Subject']}&text=@{int(length(body('GetDocuments')))} Documents Found

...



Condition

If yes
Add an action

Create file

FOLDER PATH*
/

FILE NAME*
Patient_@{guid()}.json

FILE CONTENT*
Body

Connected to docdb. Change connection.

If no, do nothing
Add an action

+

When editing the actions in the Logic App Designer, you have the option of selecting **Outputs** from the HTTP Request or from the previous action as shown in the sendMail action below.

// sendMail

METHOD*

POST

URI*

https://api.sendgrid.com/api/mail.send.json

You can insert data from previous steps...

Outputs from When an HTTP request is received

Outputs from Create file

Outputs from GetUtcDate

☐ Body

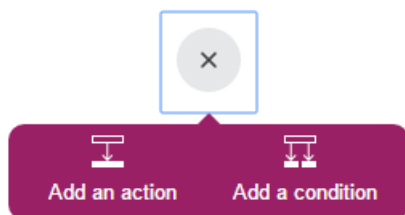
HEADERS

{"Content-type":"application/x-www-form-urlencoded"}

BODY

api_user=@{triggerBody()["sendgridUsername"]}&api_key=@{tri

Before each action in your workflow, you can make a decision; **Add an action** or **Add a condition** as shown in the following figure.



If you select **Add a condition**, you are presented with a form, as shown in the following figure, to enter your logic. This is in essence, a business rule. If you click inside a field, you have a choice of selecting parameters from the previous action. You can also enter the values directly.

↓

Condition

CONDITION

```
@greater(length(body('GetDocuments')), 0)
```

If yes Add an action

Create file
...

If no, do nothing Add an action

NOTE

You also have the capability to enter everything in Code View.

Let's take a look at the completed Logic App in code view.

```
"$schema": "https://schema.management.azure.com/providers/Microsoft.Logic/schemas/2015-08-01-preview/workflowdefinition.json#",
"actions": {
  "Conversion": {
    "Conversion": {
      "conditions": [
        {
          "dependsOn": "GetUtcDate"
        }
      ],
      "inputs": {
        "method": "post",
        "queries": {
          "currentdateTime": "@{body('GetUtcDate')}}"
        },
        "uri": "https://docdbnotificationapi-debug.azurewebsites.net/api/Conversion"
      },
      "metadata": {
        "apiDefinitionUrl": "https://docdbnotificationapi-debug.azurewebsites.net/swagger/docs/v1",
        "swaggerSource": "custom"
      },
      "type": "Http"
    },
    "Createfile": {
      "conditions": [
        {
          "expression": "@greater(length(body('GetDocuments')), 0)"
        },
        {
          "dependsOn": "GetDocuments"
        }
      ],
      "inputs": {
        "body": "@body('GetDocuments')",
        "host": {
          "api": {
            "runtimeUrl": "https://logic-apis-westus.azure-apim.net/apim/azureblob"
          }
        }
      }
    }
  }
}
```

```

        },
        "connection": {
            "name": "@parameters('$connections')['azureblob']['connectionId']"
        }
    },
    "method": "post",
    "path": "/datasets/default/files",
    "queries": {
        "folderPath": "/patients",
        "name": "Patient_@{guid()}.json"
    }
},
"type": "ApiConnection"
},
"GetDocuments": {
    "conditions": [
        {
            "dependsOn": "Conversion"
        }
    ],
    "inputs": {
        "method": "post",
        "queries": {
            "unixTimeStamp": "@body('Conversion')"
        },
        "uri": "https://docdbnotificationapi-debug.azurewebsites.net/api/Patient"
    },
    "metadata": {
        "apiDefinitionUrl": "https://docdbnotificationapi-debug.azurewebsites.net/swagger/docs/v1",
        "swaggerSource": "custom"
    },
    "type": "Http"
},
"GetUtcDate": {
    "conditions": [],
    "inputs": {
        "method": "get",
        "queries": {
            "hoursBack": "@{int(triggerBody()['GetUtcDate_HoursBack'])}"
        },
        "uri": "https://docdbnotificationapi-debug.azurewebsites.net/api/Authorization"
    },
    "metadata": {
        "apiDefinitionUrl": "https://docdbnotificationapi-debug.azurewebsites.net/swagger/docs/v1",
        "swaggerSource": "custom"
    },
    "type": "Http"
},
"sendMail": {
    "conditions": [
        {
            "dependsOn": "GetDocuments"
        }
    ],
    "inputs": {
        "body": "api_user=@{triggerBody()['sendgridUsername']}&api_key=@{triggerBody()['sendgridPassword']}&from=@{parameters('fromAddress')}&to=@{triggerBody()['EmailTo']}&subject=@{triggerBody()['Subject']}&text=@{int(length(body('GetDocuments')))} Documents Found",
        "headers": {
            "Content-type": "application/x-www-form-urlencoded"
        },
        "method": "POST",
        "uri": "https://api.sendgrid.com/api/mail.send.json"
    },
    "type": "Http"
}
},
"contentVersion": "1.0.0.0",
"outputs": {

```

```

    "Results": {
      "type": "String",
      "value": "@{int(length(body('GetDocuments')))} Records Found"
    }
  },
  "parameters": {
    "$connections": {
      "defaultValue": {},
      "type": "Object"
    },
    "fromAddress": {
      "defaultValue": "user@msn.com",
      "type": "String"
    },
    "toAddress": {
      "defaultValue": "XXXXX@XXXXXXX.net",
      "type": "String"
    }
  },
  "triggers": {
    "manual": {
      "inputs": {
        "schema": {
          "properties": {},
          "required": [],
          "type": "object"
        }
      },
      "type": "Manual"
    }
  }
}

```

If you are not familiar with what the different sections in the code represents, you can view the [Logic App Workflow Definition Language](#) documentation.

For this workflow you are using an [HTTP Webhook Trigger](#). If you look at the code above, you will see parameters like the following example.

```

=@{triggerBody()['Subject']}

```

The `triggerBody()` represents the parameters that are included in the body of an REST POST to the Logic App REST API. The `()['Subject']` represents the field. All these parameters make up the JSON formatted body.

NOTE

By using a Web hook, you can have full access to the header and body of the trigger's request. In this application you want the body.

As mentioned previously, you can use the designer to assign parameters or do it in code view. If you do it in code view, then you define which properties require a value as shown in the following code sample.

```

"triggers": {
  "manual": {
    "inputs": {
      "schema": {
        "properties": {
          "Subject": {
            "type": "String"
          }
        }
      },
      "required": [
        "Subject"
      ],
      "type": "object"
    },
    "type": "Manual"
  }
}

```

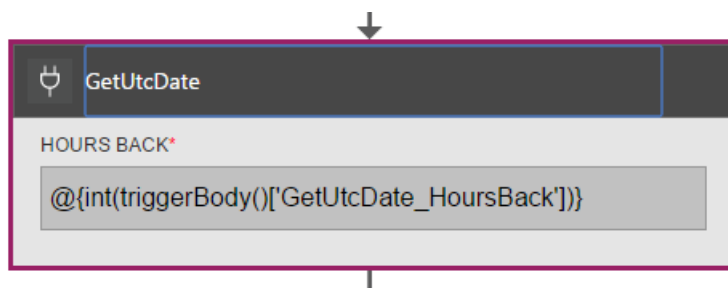
What you are doing is creating a JSON schema that will be passed in from the body of the HTTP POST. To fire your trigger, you will need a Callback URL. You will learn how to generate it later in the tutorial.

Actions

Let's see what each action in our Logic App does.

GetUTCDate

Designer View



Code View

```

"GetUtcDate": {
  "conditions": [],
  "inputs": {
    "method": "get",
    "queries": {
      "hoursBack": "@{int(triggerBody()['GetUtcDate_HoursBack'])}"
    },
    "uri": "https://docdbnotificationapi-debug.azurewebsites.net/api/Authorization"
  },
  "metadata": {
    "apiDefinitionUrl": "https://docdbnotificationapi-debug.azurewebsites.net/swagger/docs/v1"
  },
  "type": "Http"
},

```

This HTTP action performs a GET operation. It calls the API APP GetUtcDate method. The Uri uses the 'GetUtcDate_HoursBack' property passed into the Trigger body. The 'GetUtcDate_HoursBack' value is set in the first Logic App. You will learn more about the Trigger Logic App later in the tutorial.

This action calls your API App to return the UTC Date string value.

Operations

Request

```
{
  "uri": "https://docdbnotificationapi-debug.azurewebsites.net/api/Authorization",
  "method": "get",
  "queries": {
    "hoursBack": "24"
  }
}
```

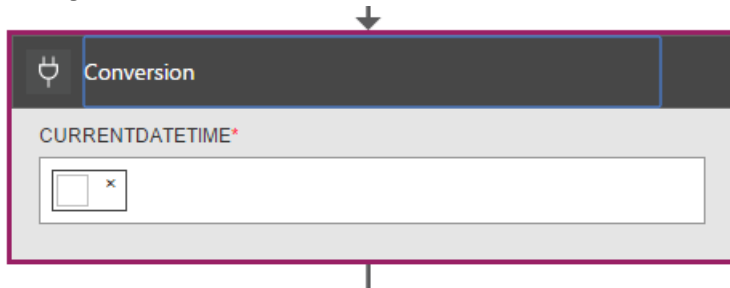
Response

```
{
  "statusCode": 200,
  "headers": {
    "pragma": "no-cache",
    "cache-Control": "no-cache",
    "date": "Fri, 26 Feb 2016 15:47:33 GMT",
    "server": "Microsoft-IIS/8.0",
    "x-AspNet-Version": "4.0.30319",
    "x-Powered-By": "ASP.NET"
  },
  "body": "Fri, 15 Jan 2016 23:47:33 GMT"
}
```

The next step is to convert the UTC DateTime value to the Unix TimeStamp, which is a .NET double type.

Conversion

D e s i g n e r V i e w



C o d e V i e w

```

"Conversion": {
  "conditions": [
    {
      "dependsOn": "GetUtcDate"
    }
  ],
  "inputs": {
    "method": "post",
    "queries": {
      "currentDateTime": "@{body('GetUtcDate')}}"
    },
    "uri": "https://docdbnotificationapi-debug.azurewebsites.net/api/Conversion"
  },
  "metadata": {
    "apiDefinitionUrl": "https://docdbnotificationapi-debug.azurewebsites.net/swagger/docs/v1"
  },
  "type": "Http"
},

```

In this step you pass in the value returned from the GetUTCDate. There is a dependsOn condition, which means that the GetUTCDate action must complete successfully. If not, then this action is skipped.

This action calls your API App to handle the conversion.

Operations

R e q u e s t

```

{
  "uri": "https://docdbnotificationapi-debug.azurewebsites.net/api/Conversion",
  "method": "post",
  "queries": {
    "currentDateTime": "Fri, 15 Jan 2016 23:47:33 GMT"
  }
}

```

R e s p o n s e

```

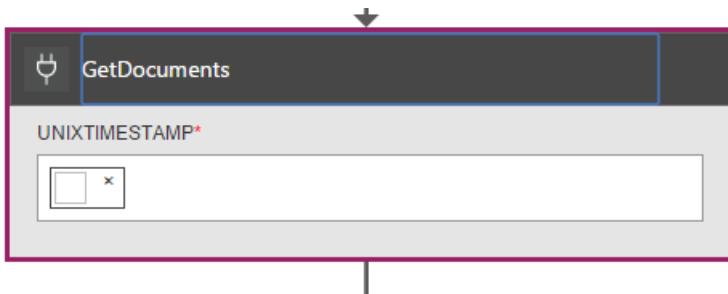
{
  "statusCode": 200,
  "headers": {
    "pragma": "no-cache",
    "cache-Control": "no-cache",
    "date": "Fri, 26 Feb 2016 15:47:33 GMT",
    "server": "Microsoft-IIS/8.0",
    "x-AspNet-Version": "4.0.30319",
    "x-Powered-By": "ASP.NET"
  },
  "body": 1452901653
}

```

In the next action, you will do a POST operation to our API App.

GetDocuments

D e s i g n e r V i e w



C o d e V i e w

```
"GetDocuments": {
  "conditions": [
    {
      "dependsOn": "Conversion"
    }
  ],
  "inputs": {
    "method": "post",
    "queries": {
      "unixTimeStamp": "@{body('Conversion')}}"
    },
    "uri": "https://docdbnotificationapi-debug.azurewebsites.net/api/Patient"
  },
  "metadata": {
    "apiDefinitionUrl": "https://docdbnotificationapi-debug.azurewebsites.net/swagger/docs/v1"
  },
  "type": "Http"
},
```

For the GetDocuments action you are going to pass in the response body from the Conversion action. This is a parameter in the Uri:

```
unixTimeStamp=@{body('Conversion')}
```

The QueryDocuments action does a HTTP POST operation to the API App.

The method called is **QueryForNewPatientDocuments**.

Operations

R e q u e s t

```
{
  "uri": "https://docdbnotificationapi-debug.azurewebsites.net/api/Patient",
  "method": "post",
  "queries": {
    "unixTimeStamp": "1452901653"
  }
}
```

R e s p o n s e

```

{
  "statusCode": 200,
  "headers": {
    "pragma": "no-cache",
    "cache-Control": "no-cache",
    "date": "Fri, 26 Feb 2016 15:47:35 GMT",
    "server": "Microsoft-IIS/8.0",
    "x-AspNet-Version": "4.0.30319",
    "x-Powered-By": "ASP.NET"
  },
  "body": [
    {
      "id": "xcda",
      "_rid": "vCYLAP2k6gAXAAAAAAAAA==",
      "_self": "dbs/vCYLAA==/colls/vCYLAP2k6gA=/docs/vCYLAP2k6gAXAAAAAAAAA==/",
      "_ts": 1454874620,
      "_etag": "\"00007d01-0000-0000-0000-56b79ffc0000\"",
      "resourceType": "Patient",
      "text": {
        "status": "generated",
        "div": "<div>\n      \n      <p>Henry Levin the 7th</p>\n      \n      </div>"
      },
      "identifier": [
        {
          "use": "usual",
          "type": {
            "coding": [
              {
                "system": "http://hl7.org/fhir/v2/0203",
                "code": "MR"
              }
            ]
          },
          "system": "urn:oid:2.16.840.1.113883.19.5",
          "value": "12345"
        }
      ],
      "active": true,
      "name": [
        {
          "family": [
            "Levin"
          ],
          "given": [
            "Henry"
          ]
        }
      ],
      "gender": "male",
      "birthDate": "1932-09-24",
      "managingOrganization": {
        "reference": "Organization/2.16.840.1.113883.19.5",
        "display": "Good Health Clinic"
      }
    }
  ],
},


```


The next action is to save the documents to [Azure Blob storage](#).

NOTE

Blob storage requires an Azure Storage account. You need to provision an Azure Blob storage account and add a new Blob named patients. For more information, see [Get started with Azure Blob storage](#).

If yes

 Add an action


 Createfile
 ...

FOLDER PATH*

...

FILE NAME*

FILE CONTENT*

 Body
 ×

Connected to docdb. [Change connection.](#)

```

{
  "host": {
    "api": {
      "runtimeUrl": "https://logic-apis-westus.azure-apim.net/apim/azureblob"
    },
    "connection": {
      "name": "subscriptions/fxxxxxc079-4e5d-b002-xxxxxxxxxx/resourceGroups/Api-Default-Central-
US/providers/Microsoft.Web/connections/azureblob"
    }
  },
  "method": "post",
  "path": "/datasets/default/files",
  "queries": {
    "folderPath": "/patients",
    "name": "Patient_17513174-e61d-4b56-88cb-5cf383db4430.json"
  },
  "body": [
    {
      "id": "xcda",
      "_rid": "vCYLAP2k6gAXAAAAAAAAA==",
      "_self": "dbs/vCYLAA==/colls/vCYLAP2k6gA=/docs/vCYLAP2k6gAXAAAAAAAAA==/",
      "_ts": 1454874620,
      "_etag": "\"00007d01-0000-0000-0000-56b79ffc0000\"",
      "resourceType": "Patient",
      "text": {
        "status": "generated",
        "div": "<div>\n      \n      <p>Henry Levin the 7th</p>\n      \n      </div>"
      },
      "identifier": [
        {
          "use": "usual",
          "type": {
            "coding": [
              {
                "system": "http://hl7.org/fhir/v2/0203",
                "code": "MR"
              }
            ]
          },
          "system": "urn:oid:2.16.840.1.113883.19.5",
          "value": "12345"
        }
      ],
      "active": true,
      "name": [
        {
          "family": [
            "Levin"
          ],
          "given": [
            "Henry"
          ]
        }
      ],
      "gender": "male",
      "birthDate": "1932-09-24",
      "managingOrganization": {
        "reference": "Organization/2.16.840.1.113883.19.5",
        "display": "Good Health Clinic"
      }
    },
  ],

```

The code is generated from action in the designer. You don't have to modify the code.

If you are not familiar with using the Azure Blob API, see [Get started with the Azure blob storage API](#).

Operations

R e q u e s t

```
"host": {
  "api": {
    "runtimeUrl": "https://logic-apis-westus.azure-apim.net/apim/azureblob"
  },
  "connection": {
    "name": "subscriptions/fxxxxxc079-4e5d-b002-xxxxxxxxxx/resourceGroups/Api-Default-Central-
US/providers/Microsoft.Web/connections/azureblob"
  }
},
"method": "post",
"path": "/datasets/default/files",
"queries": {
  "folderPath": "/patients",
  "name": "Patient_17513174-e61d-4b56-88cb-5cf383db4430.json"
},
"body": [
  {
    "id": "xcda",
    "_rid": "vCYLAP2k6gAXAAAAAAAAA==",
    "_self": "dbs/vCYLAA==/colls/vCYLAP2k6gA=/docs/vCYLAP2k6gAXAAAAAAAAA==/",
    "_ts": 1454874620,
    "_etag": "\"00007d01-0000-0000-0000-56b79ffc0000\"",
    "resourceType": "Patient",
    "text": {
      "status": "generated",
      "div": "<div>\n      \n      <p>Henry Levin the 7th</p>\n      \n      </div>"
    },
    "identifier": [
      {
        "use": "usual",
        "type": {
          "coding": [
            {
              "system": "http://hl7.org/fhir/v2/0203",
              "code": "MR"
            }
          ]
        },
        "system": "urn:oid:2.16.840.1.113883.19.5",
        "value": "12345"
      }
    ],
    "active": true,
    "name": [
      {
        "family": [
          "Levin"
        ],
        "given": [
          "Henry"
        ]
      }
    ],
    "gender": "male",
    "birthDate": "1932-09-24",
    "managingOrganization": {
      "reference": "Organization/2.16.840.1.113883.19.5",
      "display": "Good Health Clinic"
    }
  },...
]
```

R e s p o n s e

```

{
  "statusCode": 200,
  "headers": {
    "pragma": "no-cache",
    "x-ms-request-id": "2b2f7c57-2623-4d71-8e53-45c26b30ea9d",
    "cache-Control": "no-cache",
    "date": "Fri, 26 Feb 2016 15:47:36 GMT",
    "set-Cookie":
"ARRAffinity=29e552cea7db23196f7ffa644003eaf39bc8eb6dd555511f669d13ab7424faf;Path=/;Domain=127.0.0.1",
    "server": "Microsoft-HTTPAPI/2.0",
    "x-AspNet-Version": "4.0.30319",
    "x-Powered-By": "ASP.NET"
  },
  "body": {
    "Id": "0B0nBzHyMV-_NRGRDcDNMSFAxWFE",
    "Name": "Patient_47a2a0dc-640d-4f01-be38-c74690d085cb.json",
    "DisplayName": "Patient_47a2a0dc-640d-4f01-be38-c74690d085cb.json",
    "Path": "/Patient/Patient_47a2a0dc-640d-4f01-be38-c74690d085cb.json",
    "LastModified": "2016-02-26T15:47:36.215Z",
    "Size": 65647,
    "MediaType": "application/octet-stream",
    "IsFolder": false,
    "ETag": "\"c-g_a-10taH-kNQ4wBoXlp3Zv9s/MTQ1NjUwMTY1NjIxNQ\"",
    "FileLocator": "0B0nBzHyMV-_NRGRDcDNMSFAxWFE"
  }
}

```

Your last step is to send an email notification

sendEmail

Designer View

// sendMail ...

METHOD*

POST

URI*

https://api.sendgrid.com/api/mail.send.json

HEADERS

{"Content-type": "application/x-www-form-urlencoded"}

BODY

api_user=@{triggerBody()['sendgridUsername']}&api_key=@{triggerBody()['sendgridPassword']}&from=@{parameters('fromAddress')}&to=@{triggerBody()['EmailTo']}&subject=@{triggerBody()['Subject']}&text=@{int(length(body('GetDocuments')))} Documents Found

...

Code View

```

"sendMail": {
  "conditions": [
    {
      "dependsOn": "GetDocuments"
    }
  ],
  "inputs": {
    "body": "api_user=@{triggerBody()['sendgridUsername']}&api_key=@{triggerBody()
['sendgridPassword']}&from=@{parameters('fromAddress')}&to=@{triggerBody()['EmailTo']}&subject=@{triggerBody()
['Subject']}&text=@{int(length(body('GetDocuments')))} Documents Found",
    "headers": {
      "Content-type": "application/x-www-form-urlencoded"
    },
    "method": "POST",
    "uri": "https://api.sendgrid.com/api/mail.send.json"
  },
  "type": "Http"
}

```

In this action you send an email notification. You are using [SendGrid](#).

The code for this was generated using a template for Logic App and SendGrid that is in the [101-logic-app-sendgrid Github repository](#).

The HTTP operation is a POST.

The authorization parameters are in the trigger properties

```

},
"sendgridPassword": {
  "type": "SecureString"
},
"sendgridUsername": {
  "type": "String"
}

In addition, other parameters are static values set in the Parameters section of the Logic App. These
are:
},
"toAddress": {
  "defaultValue": "XXXX@XXXX.com",
  "type": "String"
},
"fromAddress": {
  "defaultValue": "XXX@msn.com",
  "type": "String"
},
"emailBody": {
  "defaultValue": "@{string(concat(int(length(actions('QueryDocuments').outputs.body)) Records
Found), '/n', actions('QueryDocuments').outputs.body)}",
  "type": "String"
},

```

The emailBody is concatenating the number of documents returned from the query, which can be "0" or more, along with, "Records Found". The rest of the parameters are set from the Trigger parameters.

This action depends on the **GetDocuments** action.

Operations

R e q u e s t

```
{
  "uri": "https://api.sendgrid.com/api/mail.send.json",
  "method": "POST",
  "headers": {
    "Content-type": "application/x-www-form-urlencoded"
  },
  "body": "api_user=azureuser@azure.com&api_key=Biz@Talk&from=user@msn.com&to=XXXX@XXXX.com&subject=New
Patients&text=37 Documents Found"
}
```

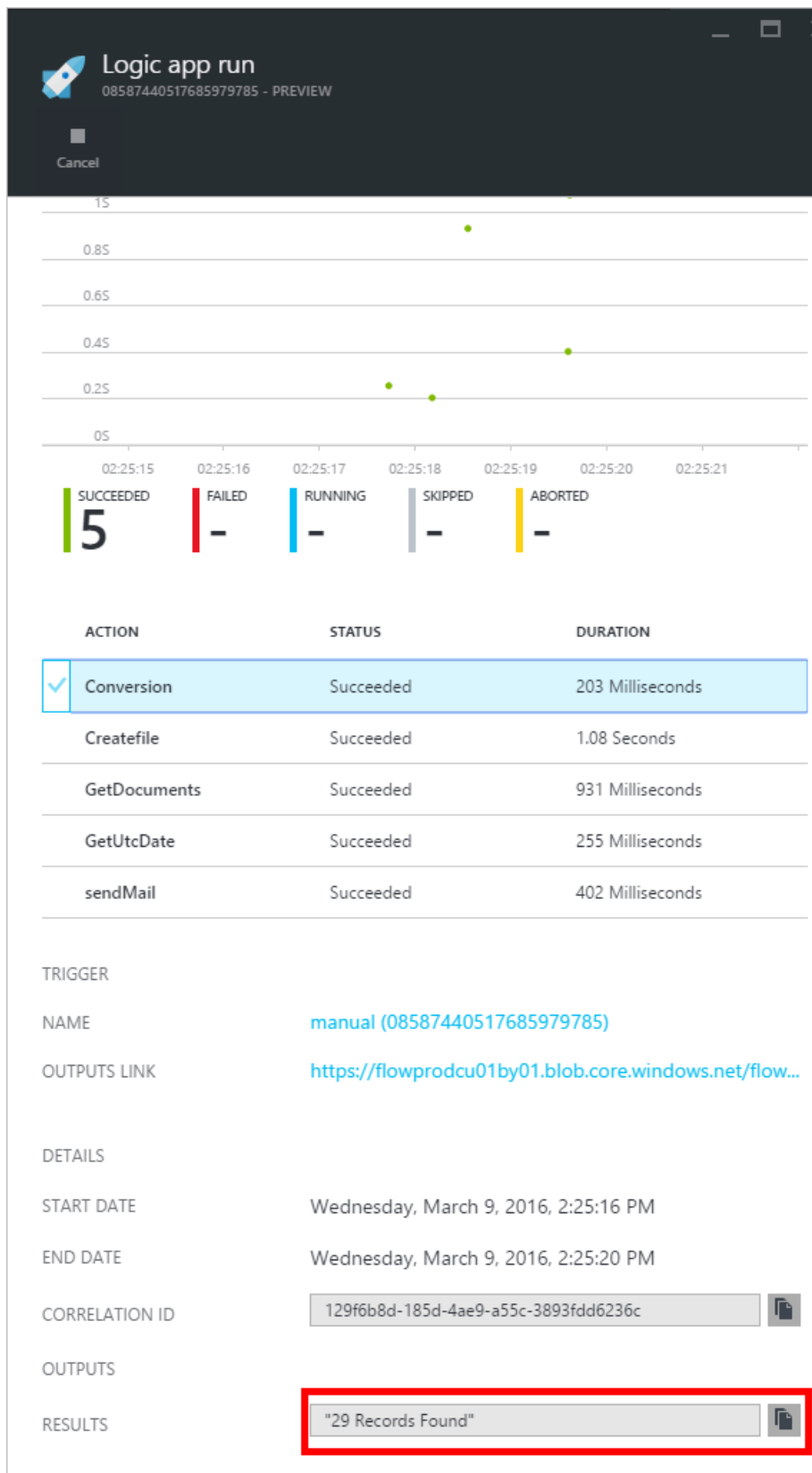
R e s p o n s e

```
{
  "statusCode": 200,
  "headers": {
    "connection": "keep-alive",
    "x-Frame-Options": "DENY,DENY",
    "access-Control-Allow-Origin": "https://sendgrid.com",
    "date": "Fri, 26 Feb 2016 15:47:35 GMT",
    "server": "nginx"
  },
  "body": {
    "message": "success"
  }
}
```

Lastly you want to be able to see the results from your Logic App on the Azure Portal. To do that, you add a parameter to the outputs section.

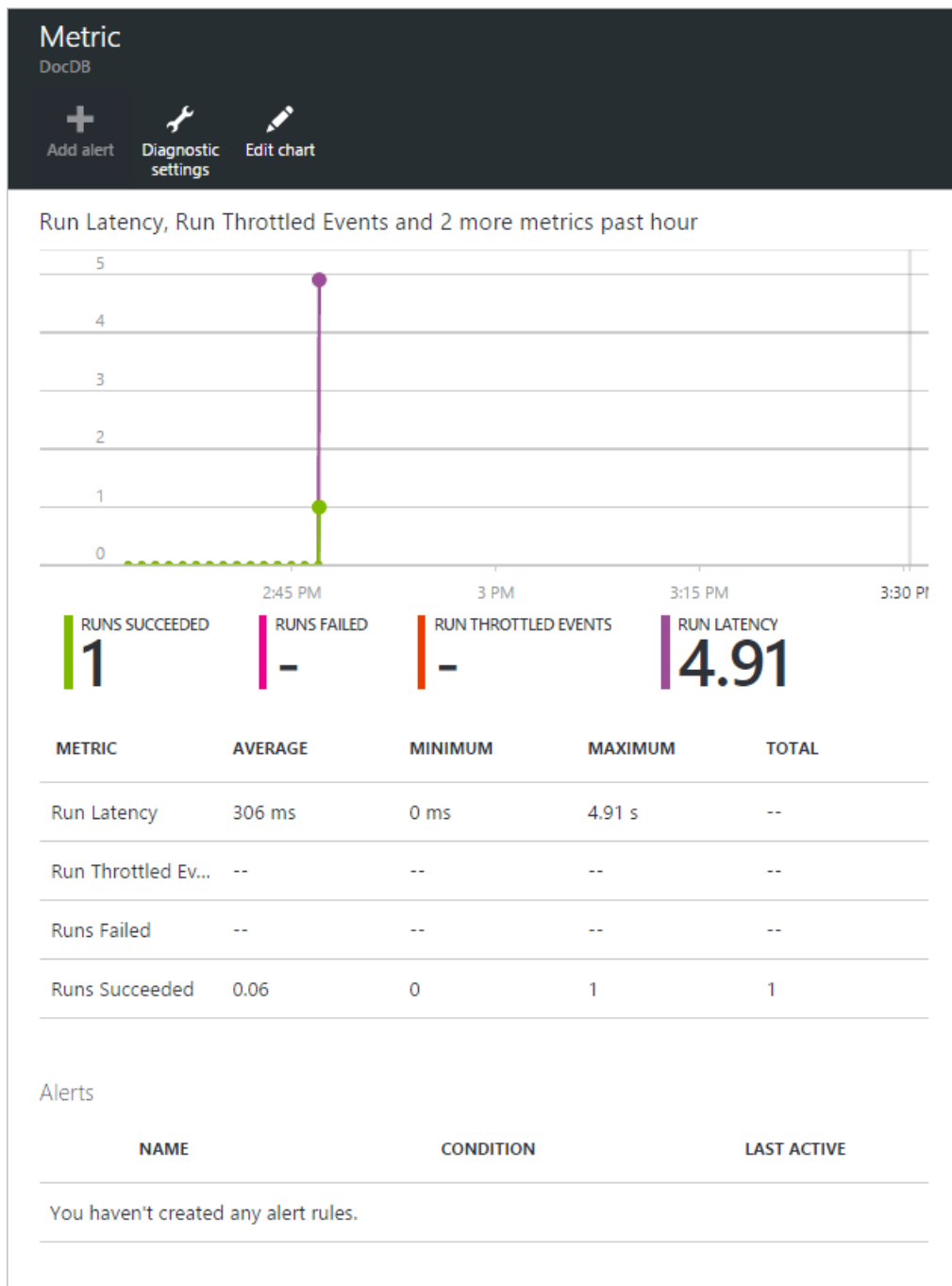
```
"outputs": {
  "Results": {
    "type": "String",
    "value": "@{int(length(actions('QueryDocuments').outputs.body))} Records Found"
  }
}
```

This returns the same value that is sent in the email body. The following figure shows an example where "29 Records Found".



Metrics

You can configure monitoring for the main Logic App in the portal. This enables you to view the Run Latency and other events as show in the following figure.



DocDb Trigger

This Logic App is the trigger that starts the workflow on your main Logic App.

The following figure shows the Designer View.



```

{
  "$schema": "https://schema.management.azure.com/providers/Microsoft.Logic/schemas/2015-08-01-preview/workflowdefinition.json#",
  "actions": {
    "Http": {
      "conditions": [],
      "inputs": {
        "body": {
          "EmailTo": "XXXXXX@XXXXX.net",
          "GetUtcDate_HoursBack": "24",
          "Subject": "New Patients",
          "sendgridPassword": "*****",
          "sendgridUsername": "azureuser@azure.com"
        },
        "method": "POST",
        "uri": "https://prod-01.westus.logic.azure.com:443/workflows/12a1de57e48845bc9ce7a247dfabc887/triggers/manual/run?api-version=2015-08-01-preview&sp=%2Ftriggers%2Fmanual%2Frun&sv=1.0&sig=ObTlihr529ATIuvuG-dhxOgBL4JZjItrvPQ8PV6973c"
      },
      "type": "Http"
    }
  },
  "contentVersion": "1.0.0.0",
  "outputs": {
    "Results": {
      "type": "String",
      "value": "@{body('Http')['status']}"
    }
  },
  "parameters": {},
  "triggers": {
    "recurrence": {
      "recurrence": {
        "frequency": "Hour",
        "interval": 24
      },
      "type": "Recurrence"
    }
  }
}

```

The Trigger is set for a recurrence of twenty-four hours. The Action is an HTTP POST that uses the Callback URL for the main Logic App. The body contains the parameters that are specified in the JSON Schema.

Operations

R e q u e s t

```

{
  "uri": "https://prod-01.westus.logic.azure.com:443/workflows/12a1de57e48845bc9ce7a247dfabc887/triggers/manual/run?api-version=2015-08-01-preview&sp=%2Ftriggers%2Fmanual%2Frun&sv=1.0&sig=ObTlihr529ATIuvuG-dhxOgBL4JZjItrvPQ8PV6973c",
  "method": "POST",
  "body": {
    "EmailTo": "XXXXXX@XXXXX.net",
    "GetUtcDate_HoursBack": "24",
    "Subject": "New Patients",
    "sendgridPassword": "*****",
    "sendgridUsername": "azureuser@azure.com"
  }
}

```

R e s p o n s e

```
{
  "statusCode": 202,
  "headers": {
    "pragma": "no-cache",
    "x-ms-ratelimit-remaining-workflow-writes": "7486",
    "x-ms-ratelimit-burst-remaining-workflow-writes": "1248",
    "x-ms-request-id": "westus:2d440a39-8ba5-4a9c-92a6-f959b8d2357f",
    "cache-Control": "no-cache",
    "date": "Thu, 25 Feb 2016 21:01:06 GMT"
  }
}
```

Now let's look at the API App.

DocDBNotificationApi

Although there are several operations in the app, you are only going to use three.

- GetUtcDate
- ConvertToTimeStamp
- QueryForNewPatientDocuments

DocDBNotificationApi Operations

Let's take a look at the Swagger documentation

NOTE

To allow you to call the operations externally, you need to add a CORS allowed origin value of "*" (without quotes) in the settings of your API App as shown in the following figure.

The screenshot shows the 'CORS' settings for the 'DocDBNotificationApi' in the Azure portal. At the top, there is a dark header with a yellow warning icon and the text 'CORS DocDBNotificationApi'. Below this are 'Save' and 'Discard' buttons. A light blue information box explains that Cross-Origin Resource Sharing (CORS) allows JavaScript code running in a browser on an external host to interact with the backend, and that origins should be specified (e.g., http://example.com:12345), with '*' used to allow all. Below the information box, the 'ALLOWED ORIGINS' section is visible. It contains a text input field with the value '*' entered, which is highlighted by a red rectangular box. To the right of the input field is a three-dot menu icon. Below this input field is another empty input field, also with a three-dot menu icon to its right.

GetUtcDate

Authorization

Show/Hide | List Operations | Expand Operations

GET

/api/Authorization

GetUtcDate

Implementation Notes

Gets the current UTC Date value minus the Hours Back

Response Class (Status 200)

Response Content Type

application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
hoursBack	<div>(required)</div>	How many hours back from the current Date Time	query	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
500	Internal Server Operation Error		
default	OK		

Try it out!

ConvertToTimeStamp

POST

/api/Conversion

Converts Universal DateTime to number

Response Class (Status 200)

Response Content Type

application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
currentdateTime	<div>(required)</div>	DateTime value to convert	query	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
400	DateTime is invalid		
500	InternalServerError		
default	OK		

Try it out!

QueryForNewPatientDocuments

POST

/api/Patient

QueryForNewDocuments

Implementation Notes

Query for new Documents where the Timestamp is greater than or equal to the DateTime value in the query parameters.

Response Class (Status 200)

OK

Model

Model Schema

{
 "Result": [
 {}
],
 "Id": 0,
 "Exception": {},
 "Status": 0,
 "IsCanceled": true,
 "IsCompleted": true,
 "CreationOptions": 0,
}

Response Content Type

application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
unixTimestamp	(required)	The DateTime value used to search from	query	double

Response Messages

HTTP Status Code	Reason	Response Model	Headers
400	The syntax of the SQL Statement is incorrect		
404	No Documents were found		
500	Internal Server Operation Error		
default	OK	<div><div>Model</div><div>Model Schema</div><div><div>{ "Result": [{}], "Id": 0, "Exception": {}, "Status": 0, "IsCanceled": true, "IsCompleted": true, "CreationOptions": 0, "AsyncState": {}, }</div></div></div>	

Try it out!

Let's take a look at the code behind this operation.

GetUtcDate

```

/// <summary>
/// Gets the current UTC Date value
/// </summary>
/// <returns></returns>
[HttpGet]
[Metadata("GetUtcDate", "Gets the current UTC Date value minus the Hours Back")]
[SwaggerOperation("GetUtcDate")]
[SwaggerResponse(HttpStatusCode.OK, type: typeof (string))]
[SwaggerResponse(HttpStatusCode.InternalServerError, "Internal Server Operation Error")]
public string GetUtcDate(
    [Metadata("Hours Back", "How many hours back from the current Date Time")] int hoursBack)
{

    return DateTime.UtcNow.AddHours(-hoursBack).ToString("r");
}

```

This operation simply returns the returns the current UTC DateTime minus the HoursBack value.

ConvertToTimeStamp

```

/// <summary>
///     Converts DateTime to double
/// </summary>
/// <param name="currentdateTime"></param>
/// <returns></returns>
[Metadata("Converts Universal DateTime to number")]
[SwaggerResponse(HttpStatusCode.OK, null, typeof (double))]
[SwaggerResponse(HttpStatusCode.BadRequest, "DateTime is invalid")]
[SwaggerResponse(HttpStatusCode.InternalServerError)]
[SwaggerOperation(nameof(ConvertToTimestamp))]
public double ConvertToTimestamp(
    [Metadata("currentdateTime", "DateTime value to convert")] string currentdateTime)
{
    double result;

    try
    {
        var uncoded = HttpContext.Current.Server.UrlDecode(currentdateTime);

        var newDateTime = DateTime.Parse(uncoded);
        //create Timespan by subtracting the value provided from the Unix Epoch
        var span = newDateTime - new DateTime(1970, 1, 1, 0, 0, 0, 0).ToLocalTime();

        //return the total seconds (which is a UNIX timestamp)
        result = span.TotalSeconds;
    }
    catch (Exception e)
    {
        throw new Exception("unable to convert to Timestamp", e.InnerException);
    }

    return result;
}

```

This operation converts the response from the GetUtcDate operation to a double value.

QueryForNewPatientDocuments

```

    /// <summary>
    ///     Query for new Patient Documents
    /// </summary>
    /// <param name="unixTimeStamp"></param>
    /// <returns>IList</returns>
    [Metadata("QueryForNewDocuments",
        "Query for new Documents where the Timestamp is greater than or equal to the DateTime value in the
query parameters."
    )]
    [SwaggerOperation("QueryForNewDocuments")]
    [SwaggerResponse(HttpStatusCode.OK, type: typeof (Task<IList<Document>>))]
    [SwaggerResponse(HttpStatusCode.BadRequest, "The syntax of the SQL Statement is incorrect")]
    [SwaggerResponse(HttpStatusCode.NotFound, "No Documents were found")]
    [SwaggerResponse(HttpStatusCode.InternalServerError, "Internal Server Operation Error")]
    // ReSharper disable once ConsiderUsingAsyncSuffix
    public IList<Document> QueryForNewPatientDocuments(
        [Metadata("UnixTimeStamp", "The DateTime value used to search from")] double unixTimeStamp)
    {
        var context = new DocumentDbContext();
        var filterQuery = string.Format(InvariantCulture, "SELECT * FROM Patient p WHERE p._ts >= {0}",
            unixTimeStamp);
        var options = new FeedOptions {MaxItemCount = -1};

        var collectionLink = UriFactory.CreateDocumentCollectionUri(DocumentDbContext.DatabaseId,
            DocumentDbContext.CollectionId);

        var response =
            context.Client.CreateDocumentQuery<Document>(collectionLink, filterQuery,
options).AsEnumerable();

        return response.ToList();
    }

```

This operation uses the [DocumentDB .NET SDK](#) to create a document query.

```
CreateDocumentQuery<Document>(collectionLink, filterQuery, options).AsEnumerable();
```

The response from the `ConvertToTimeStamp` operation (`unixTimeStamp`) is passed in. The operation returns a List of documents, `IList<Document>`.

Previously we talked about the `CallbackURL`. In order to start the workflow in your main Logic App, you will need to call it using the `CallbackURL`.

CallbackURL

To start off, you will need your Azure AD Token. It can be difficult to get this token. I was looking for an easy method and Jeff Hollan, who is an Azure Logic App program manager, recommended using the [armclient](#) in PowerShell. You can install it following the directions provided.

The operations you want to use are Login and Call ARM API.

Login: You use the same credentials for logging in to the Azure Portal.

The Call ARM Api operation is the one that will generate your `CallBackURL`.

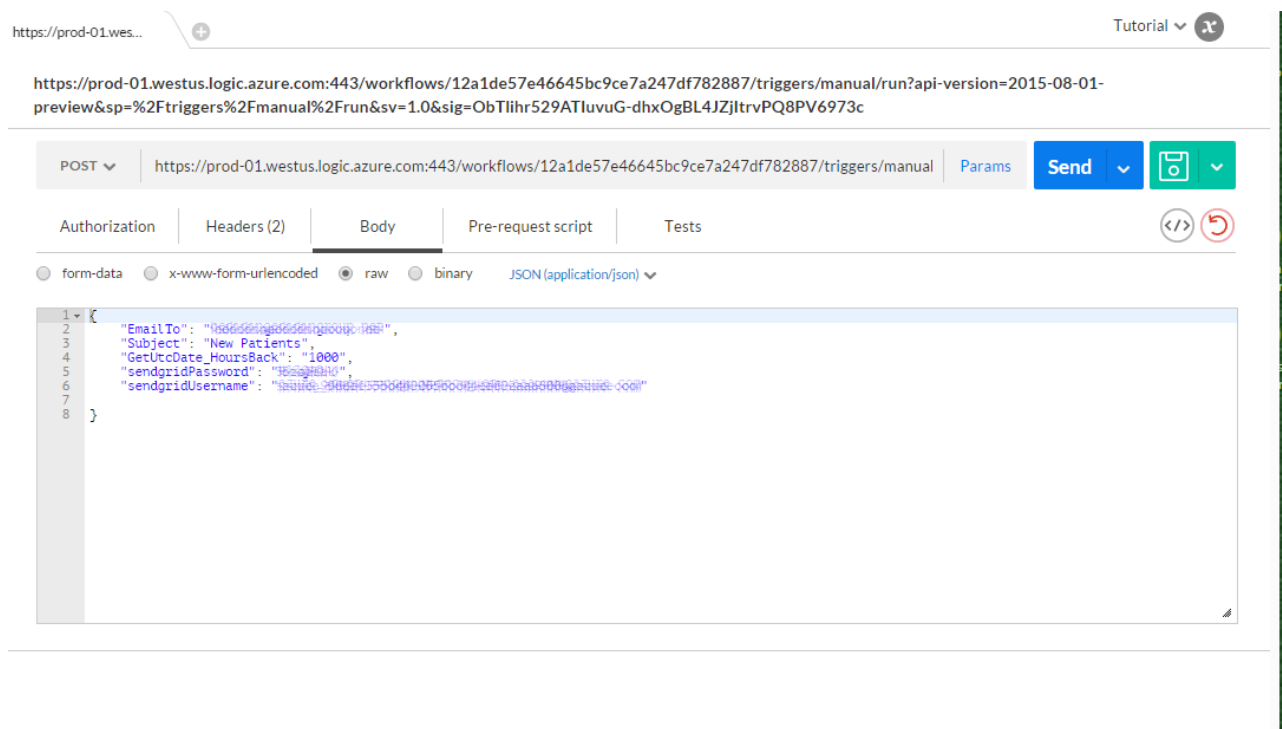
In PowerShell, you call it as follows:

```
ArmClient.exe post https://management.azure.com/subscriptions/[YOUR SUBSCRIPTION ID]/resourcegroups/[YOUR RESOURCE GROUP]/providers/Microsoft.Logic/workflows/[YOUR LOGIC APP NAME]/triggers/manual/listcallbackurl?api-version=2015-08-01-preview
```

Your result should look like this:

```
https://prod-02.westus.logic.azure.com:443/workflows/12a1de57e48845bc9ce7a247dfabc887/triggers/manual/run?api-version=2015-08-01-preview&sp=%2Ftriggers%2Fmanual%2Frun&sv=1.0&sig=XXXXXXXXXXXXXXXXXXXX
```

You can use a tool like [postman](#) to test you main Logic App as shown in the following figure.





The following table lists the Trigger parameters that make up the body of the DocDB Trigger Logic App.


PARAMETER	DESCRIPTION
GetUtcDate_HoursBack	Used to set the number of hours for the search start date
sendgridUsername	Used to set the number of hours for the search start date
sendgridPassword	The user name for Send Grid email
EmailTo	The email address that will receive the email notification
Subject	The Subject for the email

Viewing the patient data in the Azure Blob service

Go to your Azure Storage account, and select Blobs under services as shown in the following figure.

 **docdb**
Storage account

 Settings

 Delete

Essentials

Resource group
[Api-Default-Central-US](#)

Type
Standard-RAGRS

Status
Primary: Available, Secondary: Available


Location
Central US, East US 2


Subscription name
[Visual Studio Enterprise with MSDN](#)


Subscription ID
133439560741e3e11b00e521762142262


All settings

Services

 Blobs

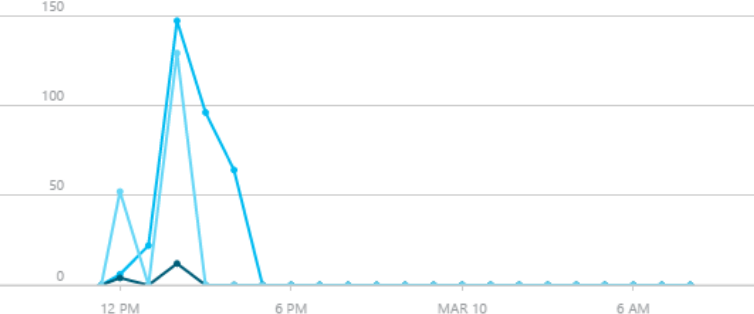
 Files

 Tables

 Queues

Monitoring

Total requests



BLOB

335

QUEUE

16

TABLE

181

Settings

docdb

Search settings

SUPPORT & TROUBLESHOOTING

Audit logs

GENERAL

Properties

Access keys

Account type

MONITORING

Alert rules

Diagnostics

RESOURCE MANAGEMENT

Users

Tags

You will be able to view the Patient blob file information as shown below.

The screenshot displays the Azure portal interface for a Blob service. It is divided into three main sections:

- Blob service (docdb):** Shows the 'Essentials' view with a search bar and a table of containers. The 'patients' container is selected, showing its URL and last modified time.
- patients (Container):** Displays a table of blobs. The first row is highlighted, showing the blob name 'Patient_02eb24d2-8bf6-421b-ae...', its modified time '3/9/2016 3:15 PM', type 'Block blob', and size '29.3 KB'.
- Blob properties:** Shows the details for the selected blob, including its name, URL, last modified time, type, size, and content-MD5.

NAME	URL	LAST MODIFIED
patients	https://docdb.blob.core.win...	3/9/2016, 2:12:22 PM

NAME	MODIFIED	BLOB TYPE	SIZE
Patient_02eb24d2-8bf6-421b-ae...	3/9/2016 3:15 PM	Block blob	29.3 KB
Patient_17513174-e61d-4b56-88...	3/9/2016 2:13 PM	Block blob	29.3 KB
Patient_1a524a02-cb05-4780-99...	3/9/2016 2:50 PM	Block blob	29.3 KB
Patient_202e581c-c17e-4aa6-89...	3/9/2016 3:20 PM	Block blob	29.3 KB
Patient_22be0a65-b355-417d-af...	3/9/2016 2:45 PM	Block blob	29.3 KB
Patient_36d35adb-a3d3-4f8f-97...	3/9/2016 3:05 PM	Block blob	29.3 KB
Patient_393dc5f3-09dc-4294-b4...	3/9/2016 4:18 PM	Block blob	29.3 KB
Patient_429183d6-1ee6-4783-87...	3/9/2016 4:20 PM	Block blob	29.3 KB
Patient_52ef79f2-d7b9-47a5-8aa...	3/9/2016 2:25 PM	Block blob	29.3 KB
Patient_658df40a-83d0-4ef3-b0...	3/9/2016 2:55 PM	Block blob	29.3 KB
Patient_668bbb93-3e7a-43fe-ba...	3/9/2016 3:00 PM	Block blob	29.3 KB
Patient_69c6ff16-c3ad-44d1-947...	3/9/2016 4:05 PM	Block blob	29.3 KB
Patient_6bab2f99-8288-4474-84...	3/9/2016 4:16 PM	Block blob	29.3 KB
Patient_725e465e-a1a4-443b-93...	3/9/2016 3:30 PM	Block blob	29.3 KB
Patient_73b4ae2e-d835-4112-80...	3/9/2016 2:40 PM	Block blob	29.3 KB
Patient_747be680-5798-4f82-b4...	3/9/2016 2:18 PM	Block blob	29.3 KB
Patient_783895be-f36d-48b1-b3...	3/9/2016 3:25 PM	Block blob	29.3 KB
Patient_7bb66e49-e8d2-4871-8c...	3/9/2016 2:35 PM	Block blob	29.3 KB

Blob properties:

- NAME: Patient_02eb24d2-8bf6-421b-aeff-04c02c434ecd.json
- URL: https://docdb.blob.core.windows.net/pati...
- LAST MODIFIED: 3/9/2016 3:15 PM
- TYPE: Block blob
- SIZE: 29.3 KB
- ETAG: 0x8D3485FEFB56912
- CONTENT-MD5: -
- LEASE STATUS: Unlocked
- LEASE STATE: Available
- LEASE DURATION: -

Summary

In this walkthrough, you've learned the following:

- It is possible to implement notifications in DocumentDB.
- By using Logic Apps, you can automate the process.
- By using Logic Apps, you can reduce the time it takes to deliver an application.
- By using HTTP you can easily consume an API App within a Logic App.
- You can easily create a CallbackURL that replaces the HTTP Listener.
- You can easily create custom workflows with Logic Apps Designer.

The key is to plan ahead and model your workflow.

Next steps

Please download and use the Logic App code provided on [Github](#). I invite you to build on the application and submit changes to the repo.

To learn more about DocumentDB, visit the [Learning Path](#).

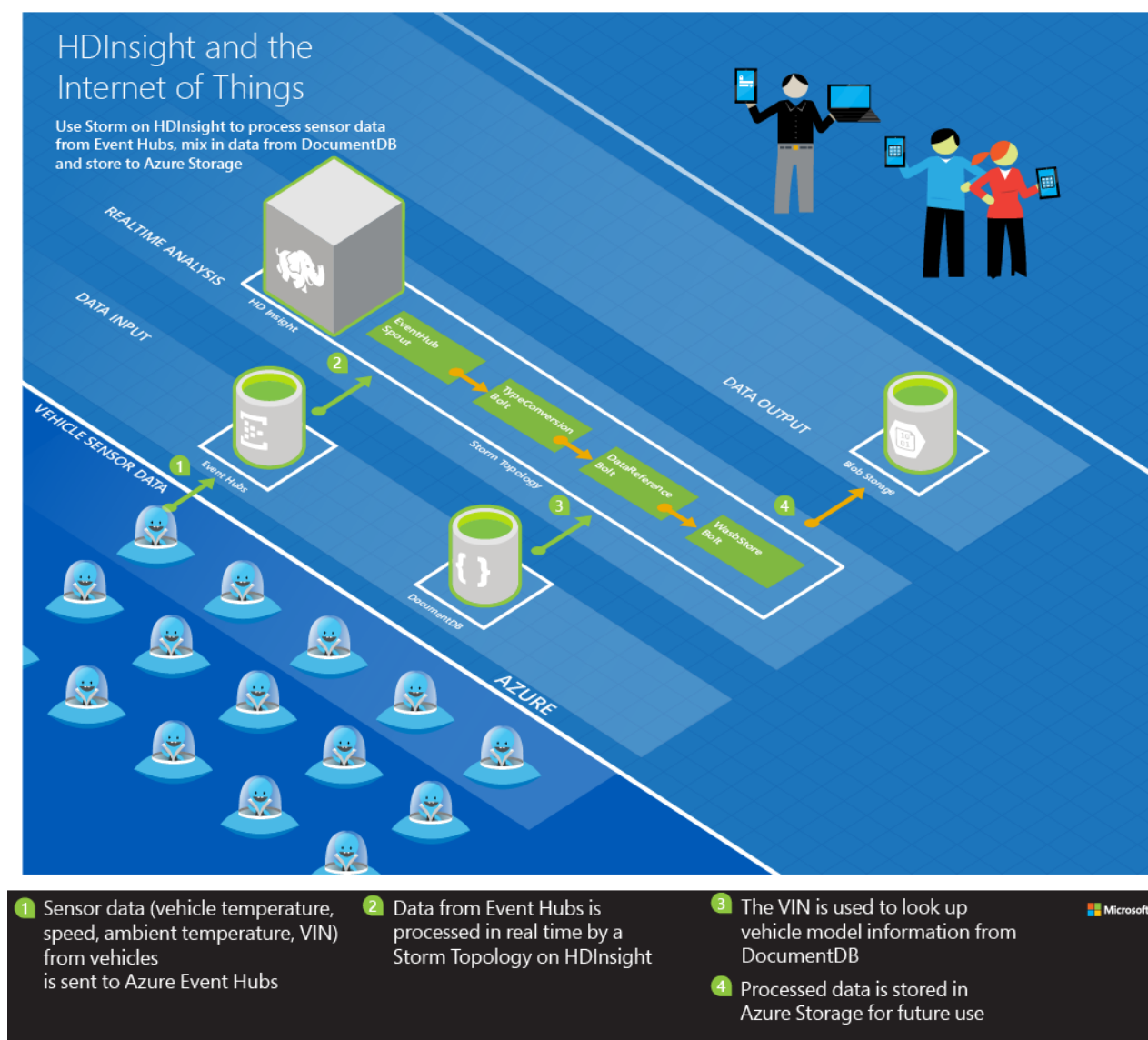
Process vehicle sensor data from Azure Event Hubs using Apache Storm on HDInsight

11/15/2016 • 2 min to read • [Edit on GitHub](#)

Contributors

Larry Franks • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Andy Pasic

Learn how to process vehicle sensor data from Azure Event Hubs using Apache Storm on HDInsight. This example reads sensor data from Azure Event Hubs, enriches the data by referencing data stored in Azure DocumentDB, and finally store the data into Azure Storage using the Hadoop File System (HDFS).



Overview

Adding sensors to vehicles allows you to predict equipment problems based on historical data trends, as well as make improvements to future versions based on usage pattern analysis. While traditional MapReduce batch processing can be used for this analysis, you must be able to quickly and efficiently load the data from all vehicles into Hadoop before MapReduce processing can occur. Additionally, you may wish to do analysis for critical failure paths (engine temperature, brakes, etc.) in real time.

Azure Event Hubs are built to handle the massive volume of data generated by sensors, and Apache Storm on HDInsight can be used to load and process the data before storing it into HDFS (backed by Azure Storage) for additional MapReduce processing.

Solution

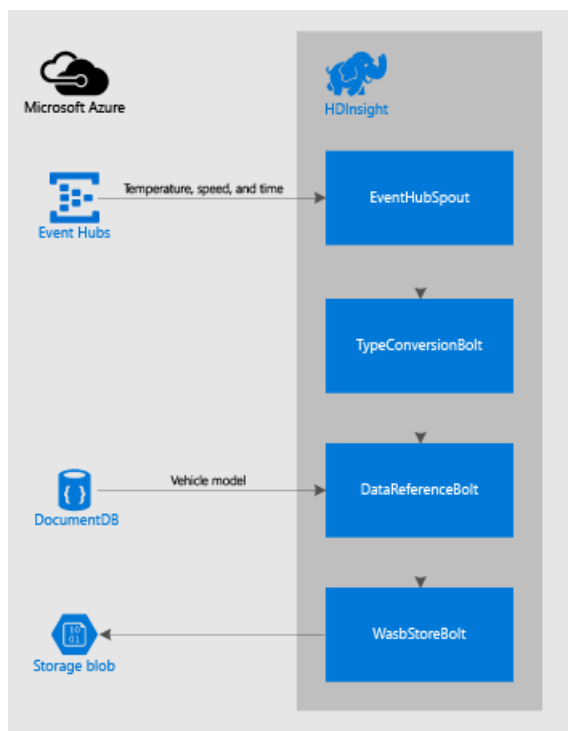
Telemetry data for engine temperature, ambient temperature, and vehicle speed is recorded by sensors, then sent to Event Hubs along with the car's Vehicle Identification Number (VIN) and a time stamp. From there, a Storm Topology running on an Apache Storm on HDInsight cluster reads the data, processes it, and stores it into HDFS.

During processing, the VIN is used to retrieve model information from Azure DocumentDB. This is added to the data stream before it is stored.

The components used in the Storm Topology are:

- **EventHubSpout** - reads data from Azure Event Hubs
- **TypeConversionBolt** - converts the JSON string from Event Hubs into a tuple containing the individual data values for engine temperature, ambient temperature, speed, VIN, and timestamp
- **DataReferencBolt** - looks up the vehicle model from DocumentDB using the VIN
- **WasbStoreBolt** - stores the data to HDFS (Azure Storage)

The following is a diagram of this solution:



NOTE

This is a simplified diagram, and each component in the solution may have multiple instances. For example, the multiple instances of each component in the topology are distributed across the nodes in the Storm on HDInsight cluster.

Implementation

A complete, automated solution for this scenario is available as part of the [HDInsight-Storm-Examples](#) repository on GitHub. To use this example, follow the steps in the [IoTExample README.MD](#).

Next Steps

For more example Storm topologies, see [Example topologies for Storm on HDInsight](#).

Power BI tutorial for DocumentDB: Visualize data using the Power BI connector

11/15/2016 • 9 min to read • [Edit on GitHub](#)

Contributors

Han Wong • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • mimig • v-aljenk

[PowerBI.com](#) is an online service where you can create and share dashboards and reports with data that's important to you and your organization. Power BI Desktop is a dedicated report authoring tool that enables you to retrieve data from various data sources, merge and transform the data, create powerful reports and visualizations, and publish the reports to Power BI. With the latest version of Power BI Desktop, you can now connect to your DocumentDB account via the DocumentDB connector for Power BI.

In this Power BI tutorial, we walk through the steps to connect to a DocumentDB account in Power BI Desktop, navigate to a collection where we want to extract the data using the Navigator, transform JSON data into tabular format using Power BI Desktop Query Editor, and build and publish a report to PowerBI.com.

After completing this Power BI tutorial, you'll be able to answer the following questions:

- How can I build reports with data from DocumentDB using Power BI Desktop?
- How can I connect to a DocumentDB account in Power BI Desktop?
- How can I retrieve data from a collection in Power BI Desktop?
- How can I transform nested JSON data in Power BI Desktop?
- How can I publish and share my reports in PowerBI.com?

Prerequisites

Before following the instructions in this Power BI tutorial, ensure that you have the following:

- [The latest version of Power BI Desktop](#).
- Access to our demo account or data in your Azure DocumentDB account.
 - The demo account is populated with the volcano data shown in this tutorial. This demo account is not bound by any SLAs and is meant for demonstration purposes only. We reserve the right to make modifications to this demo account including but not limited to, terminating the account, changing the key, restricting access, changing and delete the data, at any time without advance notice or reason.
 - URL: <https://analytics.documents.azure.com>
 - Read-only key:
MSr6kt7Gn0YRQbjd6RbTnTt7VHc5ohaAFu7osF0HdyQmfR+YhwCH2D2jcczVIR1LNK3nMPNBD31losN7IQ/fkw==
 - Or, to create your own account, see [Create a DocumentDB database account using the Azure portal](#). Then, to get sample volcano data that's similar to what's used in this tutorial (but does not contain the GeoJSON blocks), see the [NOAA site](#) and then import the data using the [DocumentDB data migration tool](#).

To share your reports in PowerBI.com, you must have an account in PowerBI.com. To learn more about Power BI for Free and Power BI Pro, please visit <https://powerbi.microsoft.com/pricing>.

Let's get started

In this tutorial, let's imagine that you are a geologist studying volcanoes around the world. The volcano data is

stored in a DocumentDB account and the JSON documents look like the one below.

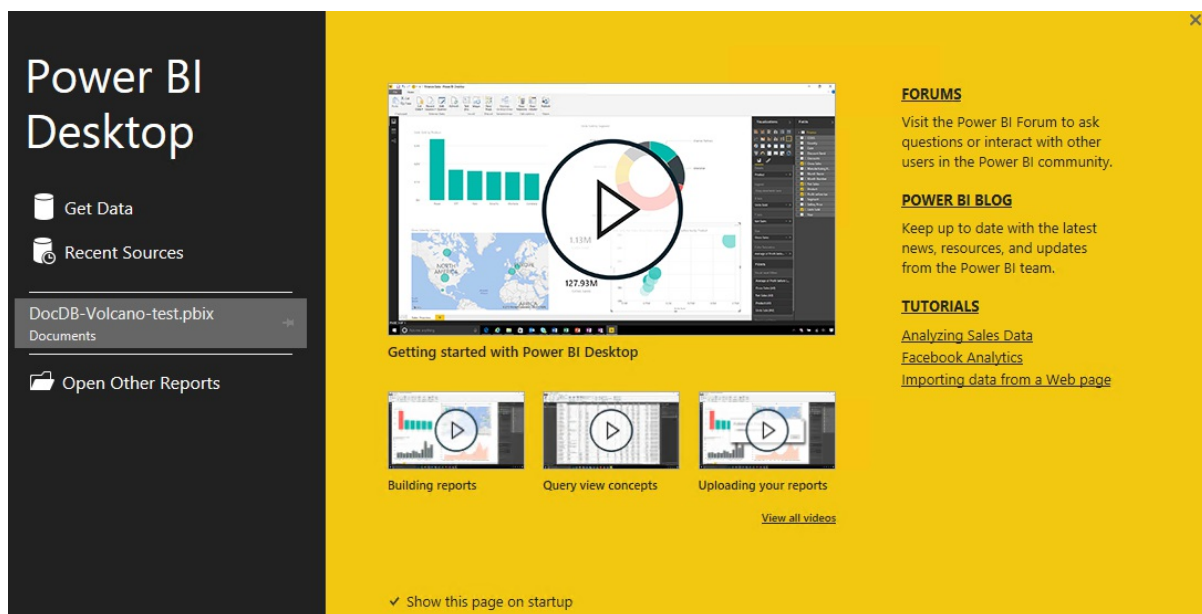
```
{
  "Volcano Name": "Rainier",
  "Country": "United States",
  "Region": "US-Washington",
  "Location": {
    "type": "Point",
    "coordinates": [
      -121.758,
      46.87
    ]
  },
  "Elevation": 4392,
  "Type": "Stratovolcano",
  "Status": "Dendrochronology",
  "Last Known Eruption": "Last known eruption from 1800-1899, inclusive"
}
```

You want to retrieve the volcano data from the DocumentDB account and visualize data in an interactive Power BI report like the one below.

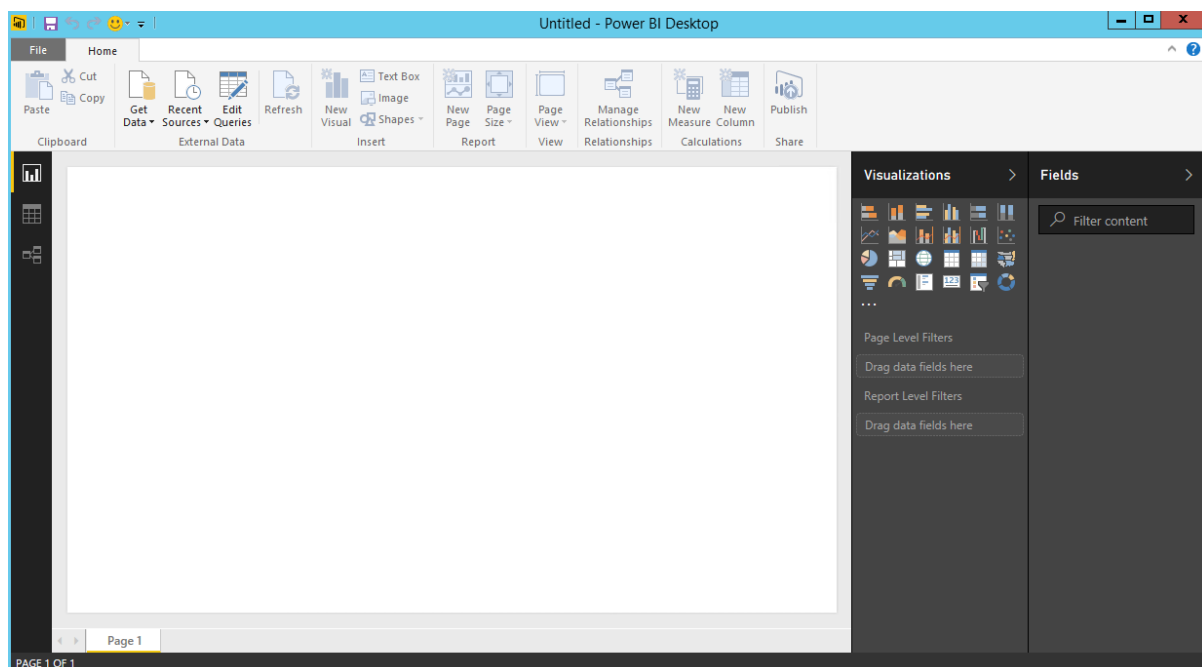


Ready to give it a try? Let's get started.

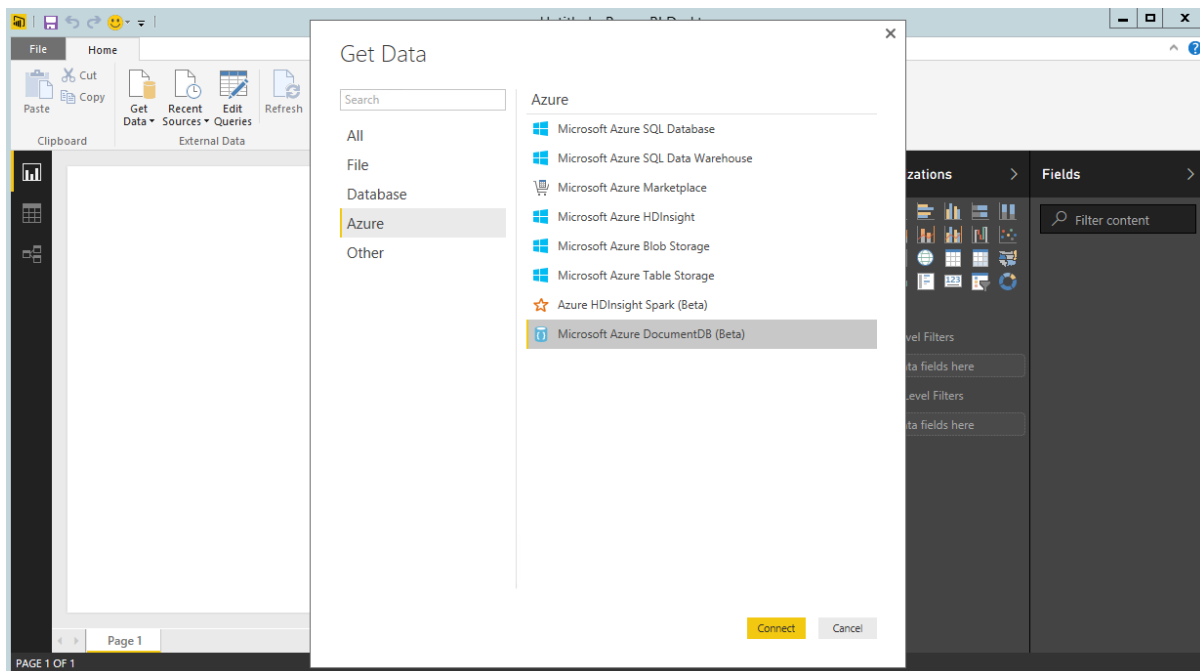
1. Run Power BI Desktop on your workstation.
2. Once Power BI Desktop is launched, a *Welcome* screen is displayed.



3. You can **Get Data**, see **Recent Sources**, or **Open Other Reports** directly from the *Welcome* screen. Click the X at the top right corner to close the screen. The **Report** view of Power BI Desktop is displayed.



4. Select the **Home** ribbon, then click on **Get Data**. The **Get Data** window should appear.
5. Click on **Azure**, select **Microsoft Azure DocumentDB (Beta)**, and then click **Connect**. The **Microsoft Azure DocumentDB Connect** window should appear.



6. Specify the DocumentDB account endpoint URL you would like to retrieve the data from as shown below, and then click **OK**. You can retrieve the URL from the **URI** box in the **Keys** blade of the Azure portal or you can use the demo account, in which case the URL is `https://analytics.documents.azure.com`.

Leave the database name, collection name, and SQL statement blank as these fields are optional. Instead, we will use the Navigator to select the Database and Collection to identify where the data comes from.

Microsoft Azure DocumentDB

Enter the URL of a Azure DocumentDB account.

URL (e.g. `https://powerquery.documents.azure.com`)

Database (optional)

Collection (optional)

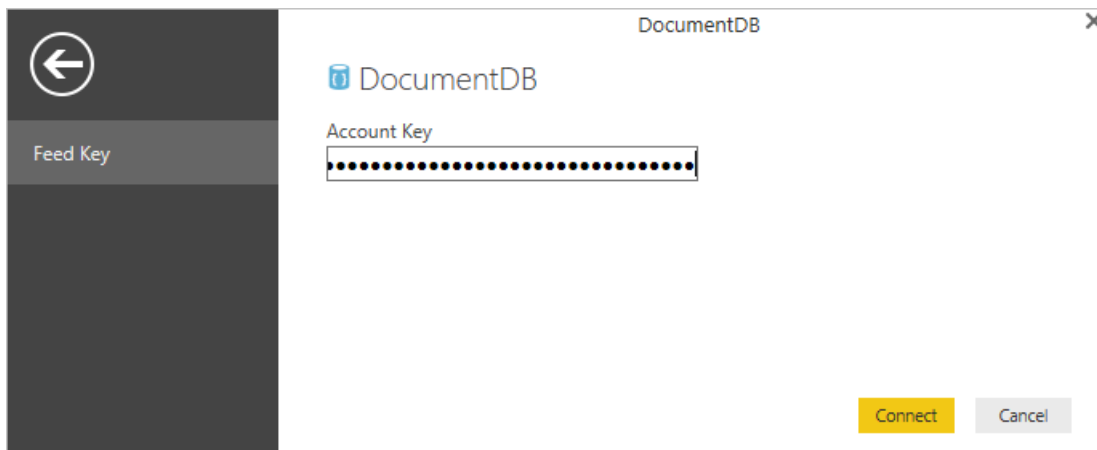
SQL statement (optional)

OK Cancel

7. If you are connecting to this endpoint for the first time, you will be prompted for the account key. You can retrieve the key from the **Primary Key** box in the **Read-only Keys** blade of the Azure portal, or you can use the demo account, in which case the key is

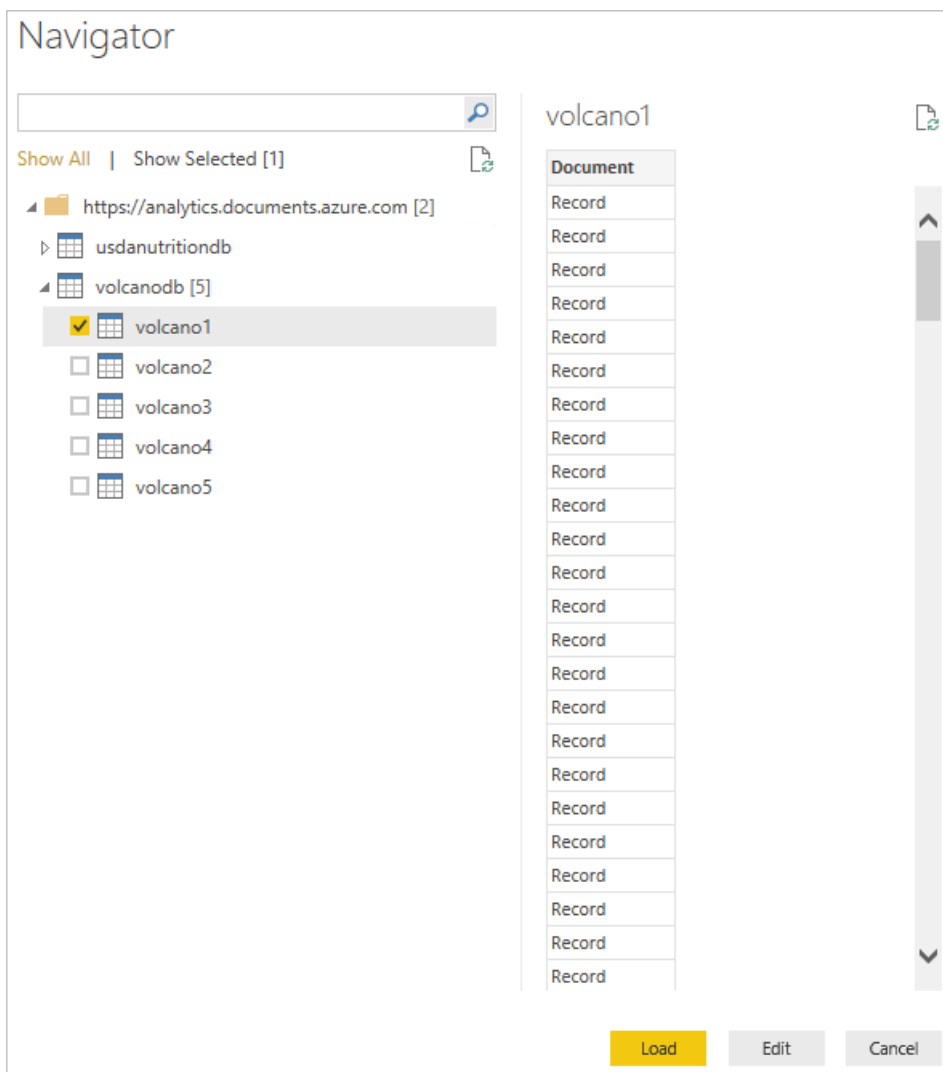
`RcEBrRI2xVn1lWheeJXncHIId6QRcKdCGQSW6uSUEgroYBWVnujW3YWvgiG2ePZ0P0TppsrMgscxs07cf6m0pcA==`. Enter the account key and click **Connect**.

We recommend that you use the read-only key when building reports. This will prevent unnecessary exposure of the master key to potential security risks. The read-only key is available from the **Keys** blade of the Azure portal or you can use the demo account information provided above.



8. When the account is successfully connected, the **Navigator** will appear. The **Navigator** will show a list of databases under the account.
9. Click and expand on the database where the data for the report will come from, if you're using the demo account, select **volcanodb**.
10. Now, select a collection that you will retrieve the data from. If you're using the demo account, select **volcano1**.

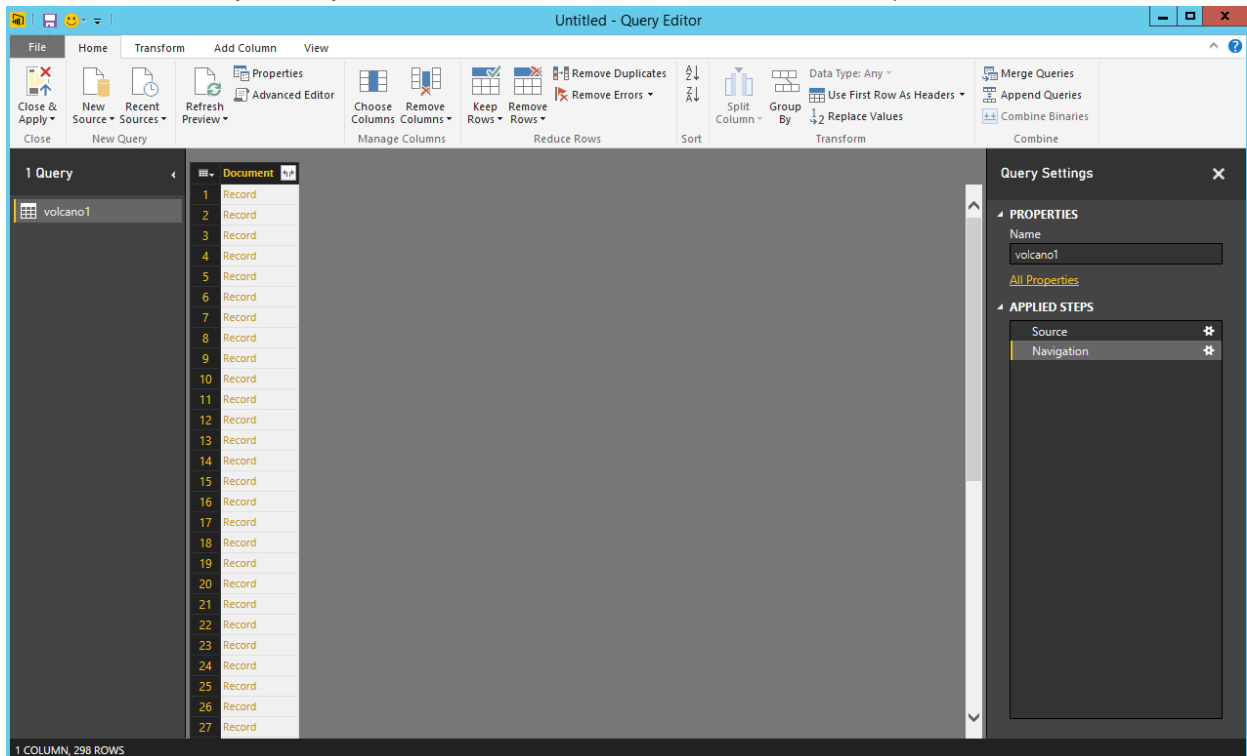
The Preview pane shows a list of **Record** items. A Document is represented as a **Record** type in Power BI. Similarly, a nested JSON block inside a document is also a **Record**.



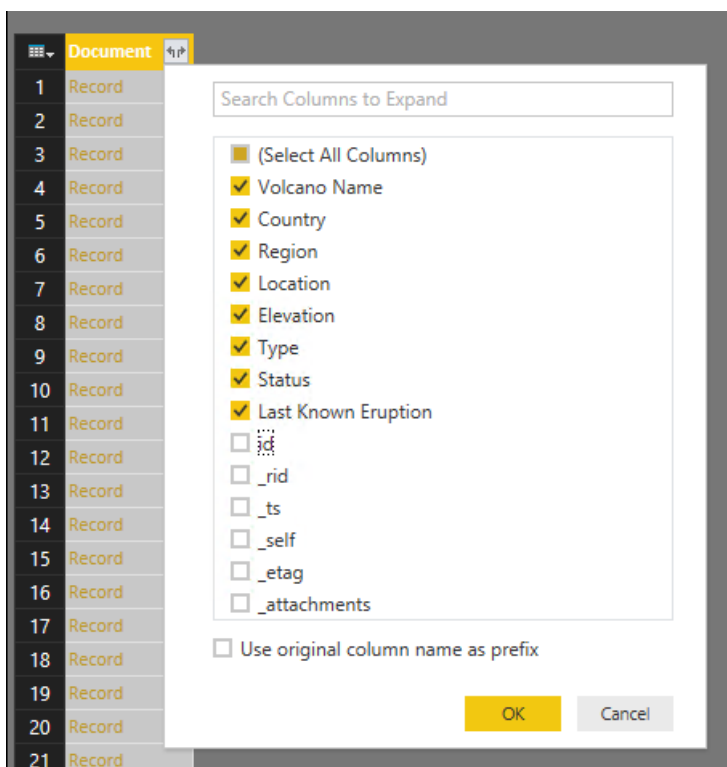
11. Click **Edit** to launch the Query Editor so we can transform the data.

Flattening and transforming JSON documents

1. In the Power BI Query Editor, you should see a **Document** column in the center pane.



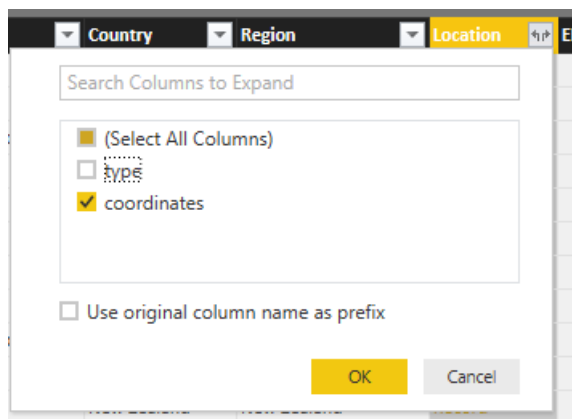
2. Click on the expander at the right side of the **Document** column header. The context menu with a list of fields will appear. Select the fields you need for your report, for instance, Volcano Name, Country, Region, Location, Elevation, Type, Status and Last Known Eruption, and then click OK.



3. The center pane will display a preview of the result with the fields selected.

	Volcano Name	Country	Region	Location	Elevation	Type	Status	Last Known Eruption
1	Razman	Iran	Iran	Record	3490	Stratovolcano	Fumarolic	Undated, but probable Holocene
2	Curtis Island	New Zealand	Kermadec Is	Record	137	Submarine volcano	Uncertain	Unknown
3	Kabargin Oth Group	Georgia	Georgia	Record	3650	Cinder cone	Holocene	Undated, but probable Holocene
4	Karapinar Field	Turkey	Turkey	Record	1302	Cinder cone	Holocene	Last known eruption B.C. (Holocene)
5	Kars Plateau	Turkey	Turkey	Record	3000	Volcanic field	Holocene	Unknown
6	La Palma	Spain	Canary Is	Record	2426	Stratovolcano	Historical	Last known eruption in 1964 or la
7	Lipari	Italy	Italy	Record	602	Stratovolcanoes	Radiocarbon	Last known eruption from A.D. 1:-
8	Lower Chindwin	Myanmar	SE Asia	Record	385	Volcanic field	Holocene	Unknown
9	Monowai Seamount	New Zealand	Kermadec Is	Record	-100	Submarine volcano	Historical	Last known eruption in 1964 or la
10	Nemrut Dagl	Turkey	Turkey	Record	3050	Stratovolcano	Historical	Last known eruption from A.D. 1:-
11	Ruapehu	New Zealand	New Zealand	Record	2797	Stratovolcano	Historical	Last known eruption in 1964 or la
12	Sete Cidades	Portugal	Azores	Record	856	Stratovolcano	Historical	Last known eruption from 1800-1
13	Tangaraa	New Zealand	New Zealand	Record	600	Submarine volcano	Fumarolic	Undated, but probable Holocene
14	Todra Volc Field	Chad	Africa-N	Record	2000	Volcanic field	Holocene	Undated, but probable Holocene
15	Unnamed	Iran	Iran	Record	null	Volcanic field	Holocene	Undated, but probable Holocene
16	Unnamed	Georgia	Georgia	Record	3400	Lava cone	Holocene	Undated, but probable Holocene
17	Voon, Tarso	Chad	Africa-N	Record	3100	Stratovolcano	Fumarolic	Undated, but probable Holocene
18	Vulcano	Italy	Italy	Record	500	Stratovolcano	Historical	Last known eruption from 1800-1
19	Aghagan-Karadag	Armenia	Armenia	Record	3560	Volcanic field	Holocene	Undated, but probable Holocene
20	Akademia Nauk	Russia	Kamchatka	Record	1180	Stratovolcanoes	Historical	Last known eruption from 1900-1
21	Bely	Russia	Kamchatka	Record	2080	Shield volcano	Holocene	Undated, but probable Holocene
22	Bezmylanny	Russia	Kamchatka	Record	2882	Stratovolcano	Historical	Last known eruption in 1964 or la
23	Biu Plateau	Nigeria	Africa-W	Record	0	Volcanic field	Holocene	Unknown
24	Cherpu Group	Russia	Kamchatka	Record	1868	Pyroclastic cones	Radiocarbon	Last known eruption B.C. (Holocene)
25	Chirinkotan	Russia	Kuril Is	Record	724	Stratovolcano	Historical	Last known eruption in 1964 or la
26	Curacoa	Tonga	Tonga-SW Pacific	Record	-33	Submarine volcano	Historical	Last known eruption in 1964 or la
27	Dar-Alages	Armenia	Armenia	Record	3329	Unknown	Holocene	Undated, but probable Holocene

- In our example, the Location property is a GeoJSON block in a document. As you can see, Location is represented as a **Record** type in Power BI Desktop.
- Click on the expander at the right side of the Location column header. The context menu with type and coordinates fields will appear. Let's select the coordinates field and click **OK**.



- The center pane now shows a coordinates column of **List** type. As shown at the beginning of the tutorial, the GeoJSON data in this tutorial is of Point type with Latitude and Longitude values recorded in the coordinates array.

The `coordinates[0]` element represents Longitude while `coordinates[1]` represents Latitude.

	Volcano Name	Country	Region	coordinates	Elevation	Type	Status	Last Known Eruption
1	Bazman	Iran	Iran	List	3490	Stratovolcano	Fumarolic	Undated, but probable Holocene
2	Curtis Island	New Zealand	Kermadec Is	List	137	Submarine volcano	Uncertain	Unknown
3	Kabargin Orh Group	Georgia	Georgia	List	3650	Cinder cone	Holocene	Undated, but probable Holocene
4	Karapinar Field	Turkey	Turkey	List	1302	Cinder cone	Holocene	Last known eruption B.C. (Holocene)
5	Kars Plateau	Turkey	Turkey	List	3000	Volcanic field	Holocene	Unknown
6	La Palma	Spain	Canary Is	List	2426	Stratovolcano	Historical	Last known eruption in 1964 o
7	Lipari	Italy	Italy	List	602	Stratovolcanoes	Radiocarbon	Last known eruption from A.D.
8	Lower Chindwin	Myanmar	SE Asia	List	385	Volcanic field	Holocene	Unknown
9	Monowai Seamount	New Zealand	Kermadec Is	List	-100	Submarine volcano	Historical	Last known eruption in 1964 o
10	Nemrut Dagı	Turkey	Turkey	List	3050	Stratovolcano	Historical	Last known eruption from A.D.
11	Ruapehu	New Zealand	New Zealand	List	2797	Stratovolcano	Historical	Last known eruption in 1964 o
12	Sete Cidades	Portugal	Azores	List	856	Stratovolcano	Historical	Last known eruption from 180
13	Tangaroa	New Zealand	New Zealand	List	600	Submarine volcano	Fumarolic	Undated, but probable Holocene
14	Todra Volc Field	Chad	Africa-N	List	2000	Volcanic field	Holocene	Undated, but probable Holocene
15	Unnamed	Iran	Iran	List	null	Volcanic field	Holocene	Undated, but probable Holocene
16	Unnamed	Georgia	Georgia	List	3400	Lava cone	Holocene	Undated, but probable Holocene
17	Voon, Tarso	Chad	Africa-N	List	3100	Stratovolcano	Fumarolic	Undated, but probable Holocene
18	Volcano	Italy	Italy	List	500	Stratovolcano	Historical	Last known eruption from 180
19	Agnagan-Karadag	Armenia	Armenia	List	3560	Volcanic field	Holocene	Undated, but probable Holocene
20	Akademik Nauk	Russia	Kamchatka	List	1180	Stratovolcanoes	Historical	Last known eruption from 190
21	Bely	Russia	Kamchatka	List	2080	Shield volcano	Holocene	Undated, but probable Holocene
22	Bezymianny	Russia	Kamchatka	List	2882	Stratovolcano	Historical	Last known eruption in 1964 o
23	Blu Plateau	Nigeria	Africa-W	List	0	Volcanic field	Holocene	Unknown
24	Cheruk Group	Russia	Kamchatka	List	1868	Pyroclastic cones	Radiocarbon	Last known eruption B.C. (Holocene)
25	Chirinkotan	Russia	Kuril Is	List	724	Stratovolcano	Historical	Last known eruption in 1964 o
26	Curacao	Tonga	Tonga-SW Pacific	List	-33	Submarine volcano	Historical	Last known eruption in 1964 o
27	Dar-Alages	Armenia	Armenia	List	3329	Unknown	Holocene	Undated, but probable Holocene

- To flatten the coordinates array, we will create a **Custom Column** called LatLong. Select the **Add Column** ribbon and click on **Add Custom Column**. The **Add Custom Column** window should appear.
- Provide a name for the new column, e.g. LatLong.
- Next, specify the custom formula for the new column. For our example, we will concatenate the Latitude and Longitude values separated by a comma as shown below using the following formula:

`Text.From([Document.Location.coordinates]{1})&", "&Text.From([Document.Location.coordinates]{0})` . Click **OK**.

For more information on Data Analysis Expressions (DAX) including DAX functions, please visit [DAX Basic in Power BI Desktop](#).

Add Custom Column

New column name

Custom column formula:

```
= Text.From([coordinates]{1})&", "&Text.From([coordinates]{0})
```

Available columns:

- Volcano Name
- Country
- Region
- coordinates
- Elevation
- Type
- Status

<< Insert

[Learn about Power BI Desktop formulas](#)

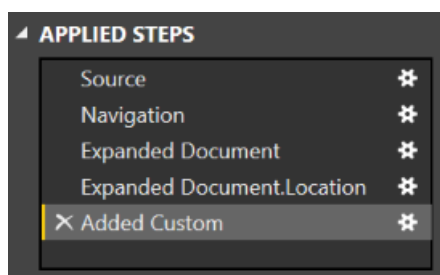
✓ No syntax errors have been detected.

OK Cancel

- Now, the center pane will show the new LatLong column populated with the Latitude and Longitude values separated by a comma.

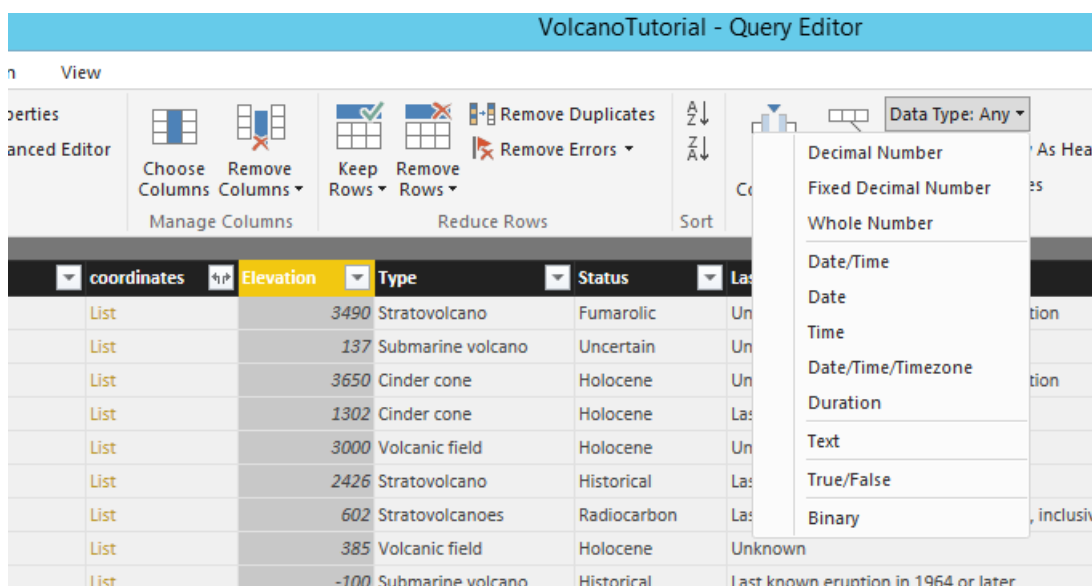
LatLong
28.07,60
-30.542,-178.561
42.55,44
37.67,33.65
40.75,42.9
28.58,-17.83
38.483,14.95
22.28,95.1
-25.887,-177.188
38.65,42.02
-39.28,175.57
37.87,-25.78
-36.321,178.028
17.683,8.5
39.25,45.167
41.55,43.6
20.92,17.28
38.404,14.962
40.275,44.75

If you receive an Error in the new column, make sure that the applied steps under Query Settings match the following figure:

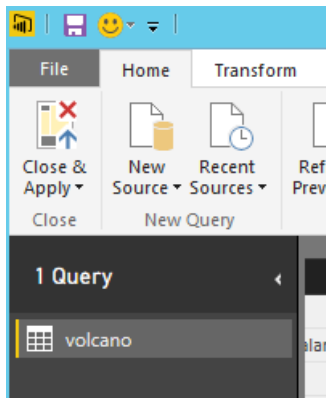


If your steps are different, delete the extra steps and try adding the custom column again.

11. We have now completed flattening the data into tabular format. You can leverage all of the features available in the Query Editor to shape and transform data in DocumentDB. If you're using the sample, change the data type for Elevation to **Whole number** by changing the **Data Type** on the **Home** ribbon.

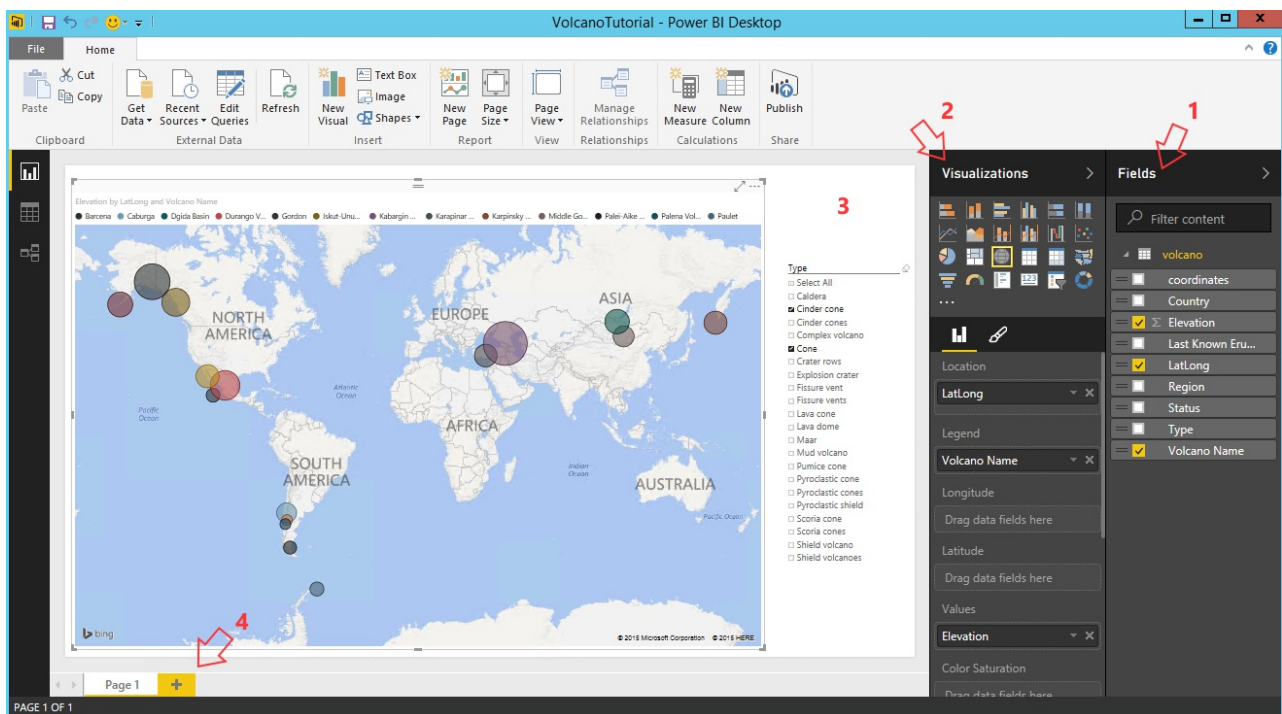


12. Click **Close and Apply** to save the data model.



Build the reports

Power BI Desktop Report view is where you can start creating reports to visualize data. You can create reports by dragging and dropping fields into the **Report** canvas.



In the Report view, you should find:

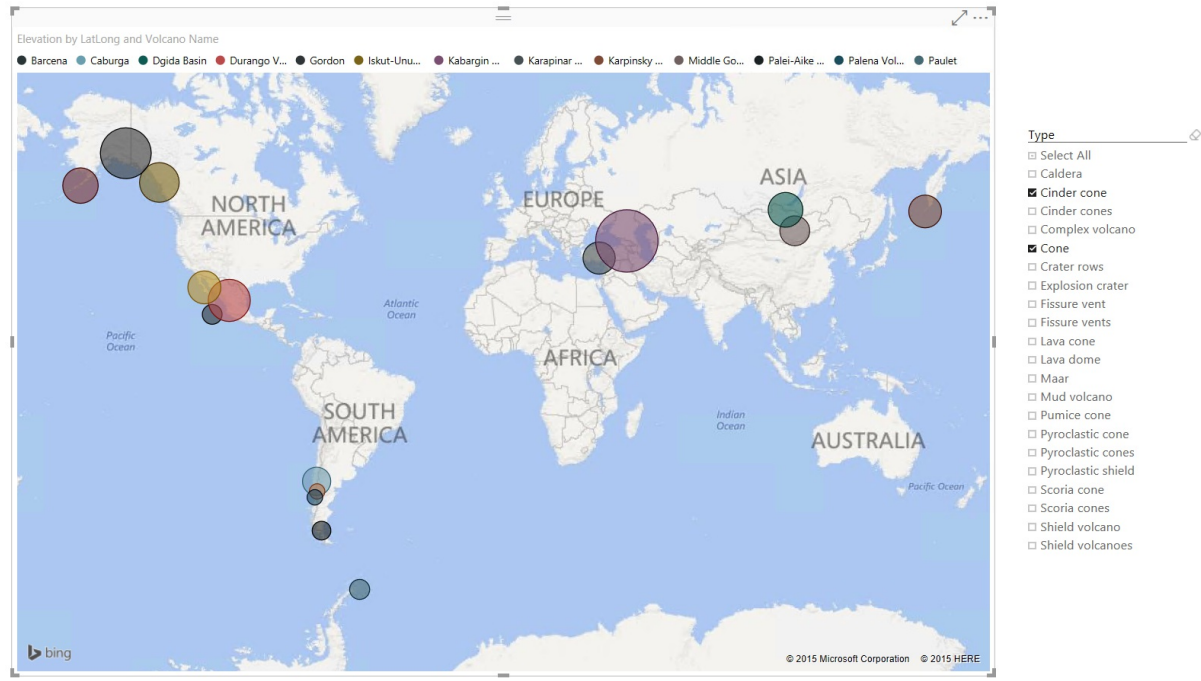
1. The **Fields** pane, this is where you will see a list of data models with fields you can use for your reports.
2. The **Visualizations** pane. A report can contain a single or multiple visualizations. Pick the visual types fitting your needs from the **Visualizations** pane.
3. The **Report** canvas, this is where you will build the visuals for your report.
4. The **Report** page. You can add multiple report pages in Power BI Desktop.

The following shows the basic steps of creating a simple interactive Map view report.

1. For our example, we will create a map view showing the location of each volcano. In the **Visualizations** pane, click on the Map visual type as highlighted in the screenshot above. You should see the Map visual type painted on the **Report** canvas. The **Visualization** pane should also display a set of properties related to the Map visual type.
2. Now, drag and drop the LatLong field from the **Fields** pane to the **Location** property in **Visualizations** pane.
3. Next, drag and drop the Volcano Name field to the **Legend** property.
4. Then, drag and drop the Elevation field to the **Size** property.
5. You should now see the Map visual showing a set of bubbles indicating the location of each volcano with the

size of the bubble correlating to the elevation of the volcano.

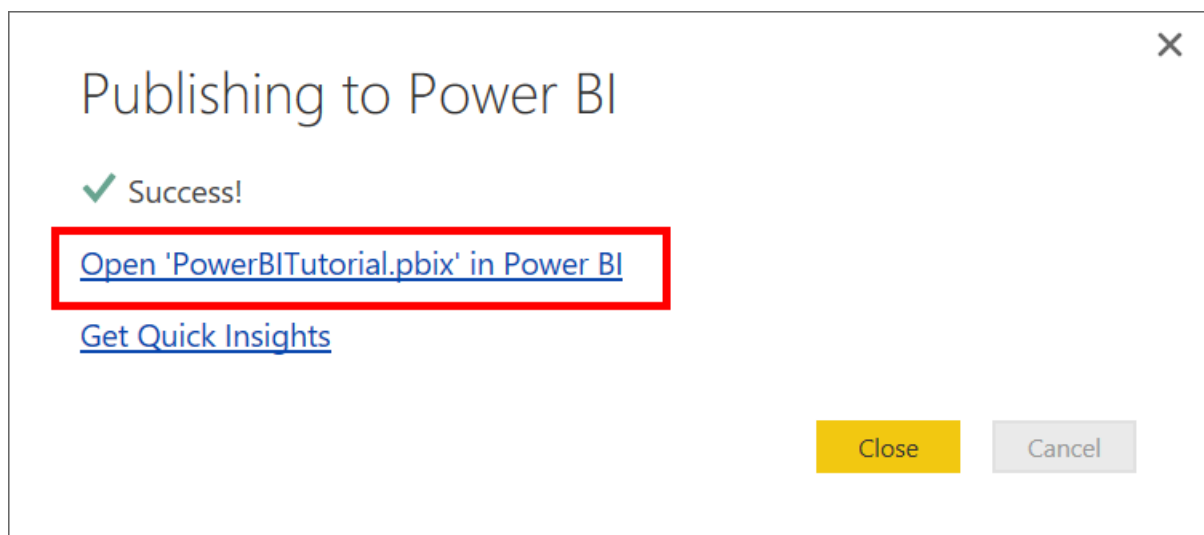
6. You now have created a basic report. You can further customize the report by adding more visualizations. In our case, we added a Volcano Type slicer to make the report interactive.



Publish and share your report

To share your report, you must have an account in PowerBI.com.

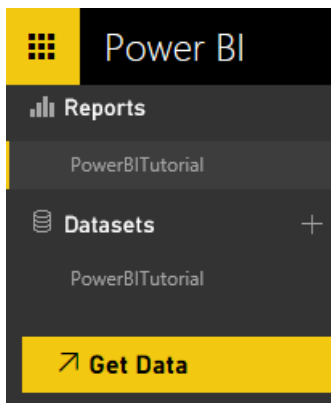
1. In the Power BI Desktop, click on the **Home** ribbon.
2. Click **Publish**. You will be prompted to enter the user name and password for your PowerBI.com account.
3. Once the credential has been authenticated, the report is published to your destination you selected.
4. Click **Open 'PowerBITutorial.pbix' in Power BI** to see and share your report on PowerBI.com.



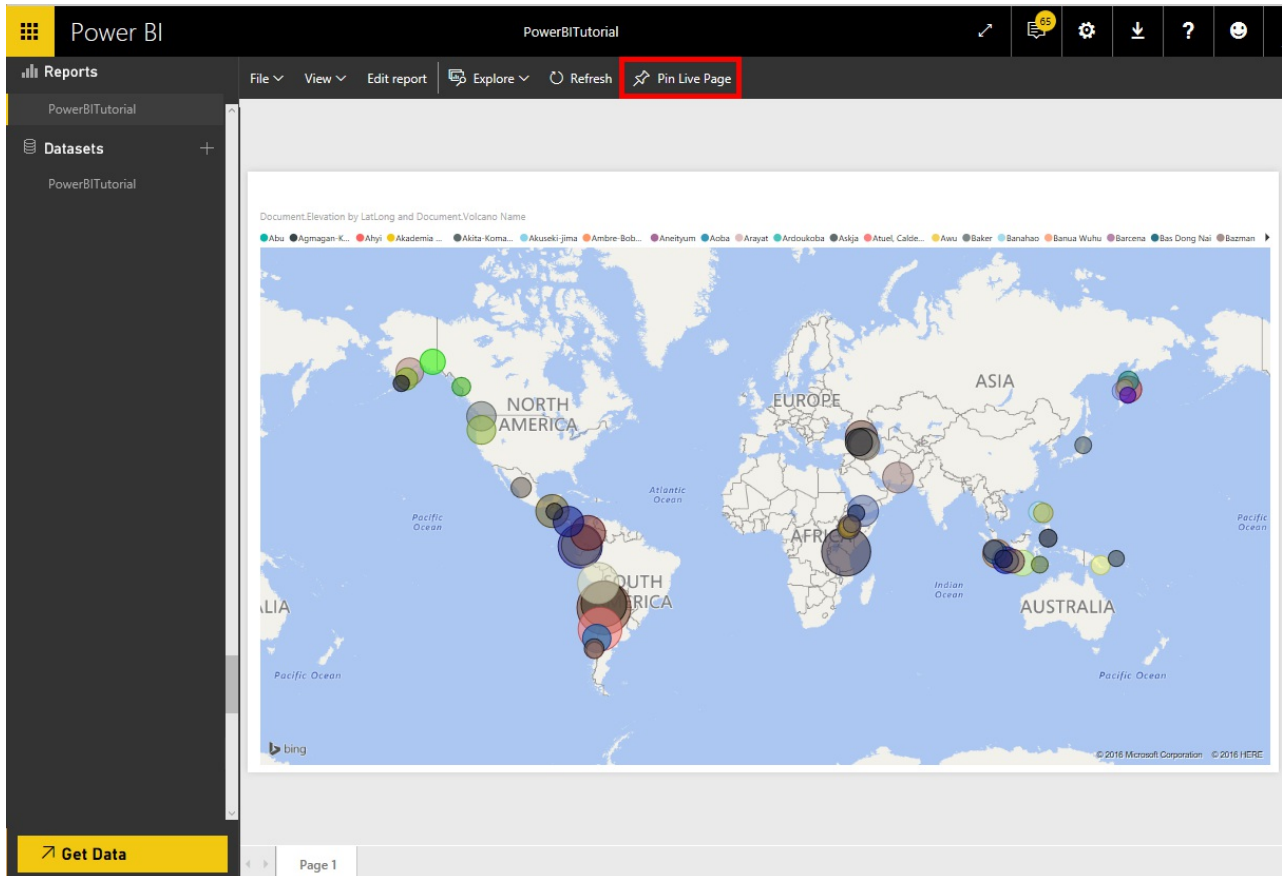
Create a dashboard in PowerBI.com

Now that you have a report, let's share it on PowerBI.com

When you publish your report from Power BI Desktop to PowerBI.com, it generates a **Report** and a **Dataset** in your PowerBI.com tenant. For example, after you published a report called **PowerBITutorial** to PowerBI.com, you will see PowerBITutorial in both the **Reports** and **Datasets** sections on PowerBI.com.



To create a sharable dashboard, click the **Pin Live Page** button on your PowerBI.com report.



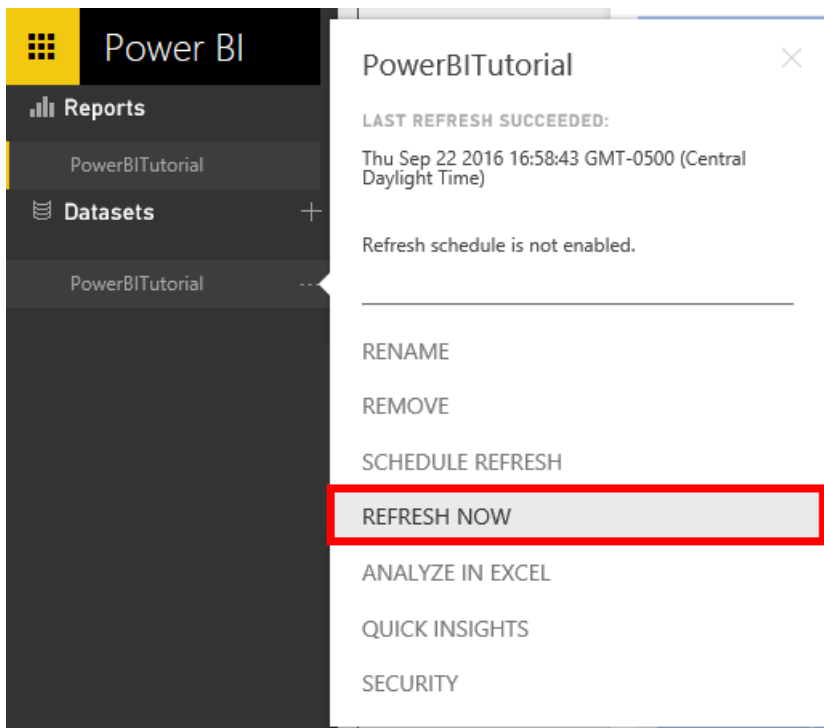
Then follow the instructions in [Pin a tile from a report](#) to create a new dashboard.

You can also do ad hoc modifications to report before creating a dashboard. However, it's recommended that you use Power BI Desktop to perform the modifications and republish the report to PowerBI.com.

Refresh data in PowerBI.com

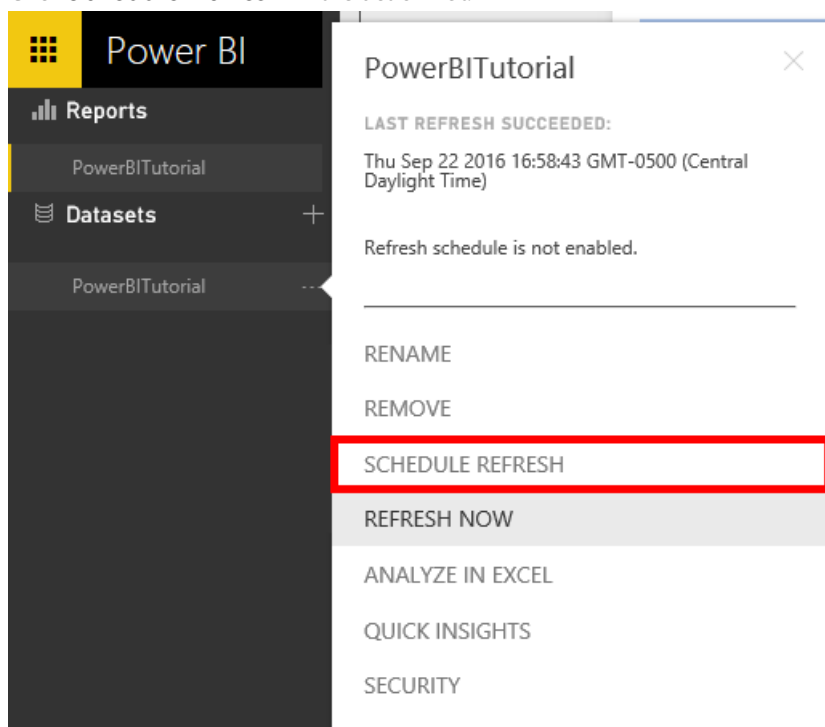
There are two ways to refresh data, ad hoc and scheduled.

For an ad hoc refresh, simply click on the eclipses (...) by the **Dataset**, e.g. PowerBITutorial. You should see a list of actions including **Refresh Now**. Click **Refresh Now** to refresh the data.



For a scheduled refresh, do the following.

1. Click **Schedule Refresh** in the action list.



2. In the **Settings** page, expand **Data source credentials**.
3. Click on **Edit credentials**.

The Configure popup appears.

4. Enter the key to connect to the DocumentDB account for that data set, then click **Sign in**.
5. Expand **Schedule Refresh** and set up the schedule you want to refresh the dataset.
6. Click **Apply** and you are done setting up the scheduled refresh.

Next steps

- To learn more about Power BI, see [Get started with Power BI](#).
- To learn more about DocumentDB, see the [DocumentDB documentation landing page](#).

DocumentDB APIs and SDKs

11/22/2016 • 4 min to read • [Edit on GitHub](#)

Contributors

Rajesh Nagpal • mimig • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • arramac • Mo Derakhshani • Andrew Liu
• Ryan CrawCour • Ross McAllister • Andy Pasic

DocumentDB Java API and SDK

SDK Download	Maven
API documentation	Java API reference documentation
Contribute to SDK	GitHub
Get started	Get started with the Java SDK
Current supported runtime	JDK 7

Release Notes

1.9.1

- Added support for BoundedStaleness consistency level.
- Added support for direct connectivity for CRUD operations for partitioned collections.
- Fixed a bug in querying a database with SQL.
- Fixed a bug in the session cache where session token may be set incorrectly.

1.9.0

- Added support for cross partition parallel queries.
- Added support for TOP/ORDER BY queries for partitioned collections.
- Added support for strong consistency.
- Added support for name based requests when using direct connectivity.
- Fixed to make ActivityId stay consistent across all request retries.
- Fixed a bug related to the session cache when recreating a collection with the same name.
- Added Polygon and LineString DataTypes while specifying collection indexing policy for geo-fencing spatial queries.
- Fixed issues with Java Doc for Java 1.8.

1.8.1

- Fixed a bug in PartitionKeyDefinitionMap to cache single partition collections and not make extra fetch partition key requests.
- Fixed a bug to not retry when an incorrect partition key value is provided.

1.8.0

- Added the support for multi-region database accounts.

- Added support for automatic retry on throttled requests with options to customize the max retry attempts and max retry wait time. See `RetryOptions` and `ConnectionPolicy.getRetryOptions()`.
- Deprecated `IPartitionResolver` based custom partitioning code. Please use partitioned collections for higher storage and throughput.

1.7.1

- Added retry policy support for throttling.

1.7.0

- Added time to live (TTL) support for documents.

1.6.0

- Implemented [partitioned collections](#) and [user-defined performance levels](#).

1.5.1

- Fixed a bug in `HashPartitionResolver` to generate hash values in little-endian to be consistent with other SDKs.

1.5.0

- Add Hash & Range partition resolvers to assist with sharding applications across multiple partitions.

1.4.0

- Implement Upsert. New `upsertXXX` methods added to support Upsert feature.
- Implement ID Based Routing. No public API changes, all changes internal.

1.3.0

- Release skipped to bring version number in alignment with other SDKs

1.2.0

- Supports GeoSpatial Index
- Validates id property for all resources. Ids for resources cannot contain `?, /, #, \` characters or end with a space.
- Adds new header "index transformation progress" to `ResourceResponse`.

1.1.0

- Implements V2 indexing policy

1.0.0

- GA SDK

Release & Retirement Dates

Microsoft will provide notification at least **12 months** in advance of retiring an SDK in order to smooth the transition to a newer/supported version.

New features and functionality and optimizations are only added to the current SDK, as such it is recommend that you always upgrade to the latest SDK version as early as possible.

Any request to DocumentDB using a retired SDK will be rejected by the service.

WARNING

All versions of the Azure DocumentDB SDK for Java prior to version **1.0.0** will be retired on **February 29, 2016**.

VERSION	RELEASE DATE	RETIREMENT DATE
1.9.1	October 28, 2016	---
1.9.0	October 03, 2016	---
1.8.1	June 30, 2016	---
1.8.0	June 14, 2016	---
1.7.1	April 30, 2016	---
1.7.0	April 27, 2016	---
1.6.0	March 29, 2016	---
1.5.1	December 31, 2015	---
1.5.0	December 04, 2015	---
1.4.0	October 05, 2015	---
1.3.0	October 05, 2015	---
1.2.0	August 05, 2015	---
1.1.0	July 09, 2015	---
1.0.1	May 12, 2015	---
1.0.0	April 07, 2015	---
0.9.5-prelease	Mar 09, 2015	February 29, 2016
0.9.4-prelease	February 17, 2015	February 29, 2016
0.9.3-prelease	January 13, 2015	February 29, 2016
0.9.2-prelease	December 19, 2014	February 29, 2016
0.9.1-prelease	December 19, 2014	February 29, 2016
0.9.0-prelease	December 10, 2014	February 29, 2016

FAQ

1. How will customers be notified of the retiring SDK?

Microsoft will provide 12 month advance notification to the end of support of the retiring SDK in order to facilitate a smooth transition to a supported SDK. Further, customers will be notified through various communication channels – Azure Management Portal, Developer Center, blog post, and direct communication to assigned service administrators.

2. Can customers author applications using a "to-be" retired DocumentDB SDK during the 12 month period?

Yes, customers will have full access to author, deploy and modify applications using the "to-be" retired DocumentDB SDK during the 12 month grace period. During the 12 month grace period, customers are advised to migrate to a newer supported version of DocumentDB SDK as appropriate.

3. Can customers author and modify applications using a retired DocumentDB SDK after the 12 month notification period?

After the 12 month notification period, the SDK will be retired. Any access to DocumentDB by an applications using a retired SDK will not be permitted by the DocumentDB platform. Further, Microsoft will not provide customer support on the retired SDK.

4. What happens to Customer's running applications that are using unsupported DocumentDB SDK version?

Any attempts made to connect to the DocumentDB service with a retired SDK version will be rejected.

5. Will new features and functionality be applied to all non-retired SDKs

New features and functionality will only be added to new versions. If you are using an old, non-retired, version of the SDK your requests to DocumentDB will still function as previous but you will not have access to any new capabilities.

6. What should I do if I cannot update my application before a cut-off date

We recommend that you upgrade to the latest SDK as early as possible. Once an SDK has been tagged for retirement you will have 12 months to update your application. If, for whatever reason, you cannot complete your application update within this timeframe then please contact the [DocumentDB Team](#) and request their assistance before the cutoff date.

See Also

To learn more about DocumentDB, see [Microsoft Azure DocumentDB](#) service page.

DocumentDB APIs and SDKs

11/22/2016 • 7 min to read • [Edit on GitHub](#)

Contributors

Rajesh Nagpal • mimig • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • arramac • Andrew Liu • Ryan CrawCour
• Andy Pasic

DocumentDB .NET API and SDK

SDK download	NuGet
API documentation	.NET API reference documentation
Samples	.NET code samples
Get started	Get started with the DocumentDB .NET SDK
Web app tutorial	Web application development with DocumentDB
Current supported framework	Microsoft .NET Framework 4.5

Release Notes

IMPORTANT

Starting with version 1.9.2 release, you may receive `System.NotSupportedException` when querying partitioned collections. To avoid this error, ensure that your host process is 64-bit. For Executable projects, this can be done by unchecking the "Prefer 32-bit" option in the project properties window, on the Build tab.

1.10.0

- Added direct connectivity support for partitioned collections.
- Improved performance for the Bounded Staleness consistency level.
- Added Polygon and LineString DataTypes while specifying collection indexing policy for geo-fencing spatial queries.
- Added LINQ support for `StringEnumConverter`, `IsoDateTimeConverter` and `UnixDateTimeConverter` while translating predicates.
- Various SDK bug fixes.

1.9.5

- Fixed an issue that caused the following `NotFoundException`: The read session is not available for the input session token. This exception occurred in some cases when querying for the read-region of a geo-distributed account.
- Exposed the `ResponseStream` property in the `ResourceResponse` class, which enables direct access to the underlying stream from a response.

1.9.4

- Modified the ResourceResponse, FeedResponse, StoredProcedureResponse and MediaResponse classes to implement the corresponding public interface so that they can be mocked for test driven deployment (TDD).
- Fixed an issue that caused a malformed partition key header when using a custom JsonSerializerSettings object for serializing data.

1.9.3

- Fixed an issue that caused long running queries to fail with error: Authorization token is not valid at the current time.
- Fixed an issue that removed the original SqlParameterCollection from cross partition top/order-by queries.

1.9.2

- Added support for parallel queries for partitioned collections.
- Added support for cross partition ORDER BY and TOP queries for partitioned collections.
- Fixed the missing references to DocumentDB.Spatial.Sql.dll and Microsoft.Azure.Documents.ServiceInterop.dll that are required when referencing a DocumentDB project with a reference to the DocumentDB Nuget package.
- Fixed the ability to use parameters of different types when using user-defined functions in LINQ.
- Fixed a bug for globally replicated accounts where Upsert calls were being directed to read locations instead of write locations.
- Added methods to the IDocumentClient interface that were missing:
 - UpsertAttachmentAsync method that takes mediaStream and options as parameters
 - CreateAttachmentAsync method that takes options as a parameter
 - CreateOfferQuery method that takes querySpec as a parameter.
- Unsealed public classes that are exposed in the IDocumentClient interface.

1.8.0

- Added the support for multi-region database accounts.
- Added support for retry on throttled requests. User can customize the number of retries and the max wait time by configuring the ConnectionPolicy.RetryOptions property.
- Added a new IDocumentClient interface that defines the signatures of all DocumentClient properties and methods. As part of this change, also changed extension methods that create IQueryable and IOrderedQueryable to methods on the DocumentClient class itself.
- Added configuration option to set the ServicePoint.ConnectionLimit for a given DocumentDB endpoint Uri. Use ConnectionPolicy.MaxConnectionLimit to change the default value, which is set to 50.
- Deprecated IPartitionResolver and its implementation. Support for IPartitionResolver is now obsolete. It's recommended that you use Partitioned Collections for higher storage and throughput.

1.7.1

- Added an overload to Uri based ExecuteStoredProcureAsync method that takes RequestOptions as a parameter.

1.7.0

- Added time to live (TTL) support for documents.

1.6.3

- Fixed a bug in Nuget packaging of .NET SDK for packaging it as part of an Azure Cloud Service solution.

1.6.2

- Implemented [partitioned collections](#) and [user-defined performance levels](#).

1.5.3

- **[Fixed]** Querying DocumentDB endpoint throws: 'System.Net.Http.HttpRequestException: Error while copying content to a stream.'

1.5.2

- Expanded LINQ support including new operators for paging, conditional expressions and range comparison.
 - Take operator to enable SELECT TOP behavior in LINQ
 - CompareTo operator to enable string range comparisons
 - Conditional (?) and coalesce operators (??)
- **[Fixed]** ArgumentOutOfRangeException when combining Model projection with Where-In in linq query. [#81](#)

1.5.1

- **[Fixed]** If Select is not the last expression the LINQ Provider assumed no projection and produced SELECT * incorrectly. [#58](#)

1.5.0

- Implemented Upsert, Added UpsertXXXAsync methods
- Performance improvements for all requests
- LINQ Provider support for conditional, coalesce and CompareTo methods for strings
- **[Fixed]** LINQ provider --> Implement Contains method on List to generate the same SQL as on IEnumerable and Array
- **[Fixed]** BackoffRetryUtility uses the same HttpRequestMessage again instead of creating a new one on retry
- **[Obsolete]** UriFactory.CreateCollection --> should now use UriFactory.CreateDocumentCollection

1.4.1

- **[Fixed]** Localization issues when using non en culture info such as nl-NL etc.

1.4.0

- ID Based Routing
 - New UriFactory helper to assist with constructing ID based resource links
 - New overloads on DocumentClient to take in URI
- Added IsValid() and IsValidDetailed() in LINQ for geospatial
- LINQ Provider support enhanced
 - **Math** - Abs, Acos, Asin, Atan, Ceiling, Cos, Exp, Floor, Log, Log10, Pow, Round, Sign, Sin, Sqrt, Tan, Truncate
 - **String** - Concat, Contains, EndsWith, IndexOf, Count, ToLower, TrimStart, Replace, Reverse, TrimEnd, StartsWith, SubString, ToUpper
 - **Array** - Concat, Contains, Count
 - **IN** operator

1.3.0

- Added support for modifying indexing policies
 - New ReplaceDocumentCollectionAsync method in DocumentClient
 - New IndexTransformationProgress property in ResourceResponse for tracking percent progress of index policy changes
 - DocumentCollection.IndexingPolicy is now mutable
- Added support for spatial indexing and query
 - New Microsoft.Azure.Documents.Spatial namespace for serializing/deserializing spatial types like

Point and Polygon

- New SpatialIndex class for indexing GeoJSON data stored in DocumentDB
- **[Fixed]** : Incorrect SQL query generated from linq expression [#38](#)

1.2.0

- Dependency on Newtonsoft.Json v5.0.7
- Changes to support Order By
 - LINQ provider support for OrderBy() or OrderByDescending()
 - IndexingPolicy to support Order By

****NB: Possible breaking change****

If you have existing code that provisions collections with a custom indexing policy, then your existing code will need to be updated to support the new IndexingPolicy class. If you have no custom indexing policy, then this change does not affect you.

1.1.0

- Support for partitioning data by using the new HashPartitionResolver and RangePartitionResolver classes and the IPartitionResolver
- DataContract serialization
- Guid support in LINQ provider
- UDF support in LINQ

1.0.0

- GA SDK

NOTE

There was a change of NuGet package name between preview and GA. We moved from **Microsoft.Azure.Documents.Client** to **Microsoft.Azure.DocumentDB**

0.9.x-preview

- Preview SDKs [Obsolete]

Release & Retirement Dates

Microsoft will provide notification at least **12 months** in advance of retiring an SDK in order to smooth the transition to a newer/supported version.

New features and functionality and optimizations are only added to the current SDK, as such it is recommended that you always upgrade to the latest SDK version as early as possible.

Any request to DocumentDB using a retired SDK will be rejected by the service.

WARNING

All versions of the Azure DocumentDB SDK for .NET prior to version **1.0.0** will be retired on **February 29, 2016**.

VERSION	RELEASE DATE	RETIREMENT DATE
1.10.0	September 27, 2016	---
1.9.5	September 01, 2016	---
1.9.4	August 24, 2016	---
1.9.3	August 15, 2016	---
1.9.2	July 23, 2016	---
1.9.1	Deprecated	---
1.9.0	Deprecated	---
1.8.0	June 14, 2016	---
1.7.1	May 06, 2016	---
1.7.0	April 26, 2016	---
1.6.3	April 08, 2016	---
1.6.2	March 29, 2016	---
1.5.3	February 19, 2016	---
1.5.2	December 14, 2015	---
1.5.1	November 23, 2015	---
1.5.0	October 05, 2015	---
1.4.1	August 25, 2015	---
1.4.0	August 13, 2015	---
1.3.0	August 05, 2015	---
1.2.0	July 06, 2015	---
1.1.0	April 30, 2015	---
1.0.0	April 08, 2015	---
0.9.3-prelease	March 12, 2015	February 29, 2016
0.9.2-prelease	January , 2015	February 29, 2016
.9.1-prelease	October 13, 2014	February 29, 2016

VERSION	RELEASE DATE	RETIREMENT DATE
0.9.0-prelease	August 21, 2014	February 29, 2016

FAQ

1. How will customers be notified of the retiring SDK?

Microsoft will provide 12 month advance notification to the end of support of the retiring SDK in order to facilitate a smooth transition to a supported SDK. Further, customers will be notified through various communication channels – Azure Management Portal, Developer Center, blog post, and direct communication to assigned service administrators.

2. Can customers author applications using a "to-be" retired DocumentDB SDK during the 12 month period?

Yes, customers will have full access to author, deploy and modify applications using the "to-be" retired DocumentDB SDK during the 12 month grace period. During the 12 month grace period, customers are advised to migrate to a newer supported version of DocumentDB SDK as appropriate.

3. Can customers author and modify applications using a retired DocumentDB SDK after the 12 month notification period?

After the 12 month notification period, the SDK will be retired. Any access to DocumentDB by an applications using a retired SDK will not be permitted by the DocumentDB platform. Further, Microsoft will not provide customer support on the retired SDK.

4. What happens to Customer's running applications that are using unsupported DocumentDB SDK version?

Any attempts made to connect to the DocumentDB service with a retired SDK version will be rejected.

5. Will new features and functionality be applied to all non-retired SDKs

New features and functionality will only be added to new versions. If you are using an old, non-retired, version of the SDK your requests to DocumentDB will still function as previous but you will not have access to any new capabilities.

6. What should I do if I cannot update my application before a cut-off date

We recommend that you upgrade to the latest SDK as early as possible. Once an SDK has been tagged for retirement you will have 12 months to update your application. If, for whatever reason, you cannot complete your application update within this timeframe then please contact the [DocumentDB Team](#) and request their assistance before the cutoff date.

See Also

To learn more about DocumentDB, see [Microsoft Azure DocumentDB](#) service page.

DocumentDB APIs and SDKs

11/22/2016 • 1 min to read • [Edit on GitHub](#)

Contributors

[Rajesh Nagpal](#) • [mimig](#)

DocumentDB .NET Core API and SDK

SDK download	NuGet
API documentation	.NET API reference documentation
Samples	.NET code samples
Get started	Get started with the DocumentDB .NET Core SDK
Web app tutorial	Web application development with DocumentDB
Current supported framework	.NET Standard 1.6

Release Notes

0.1.0-preview

The DocumentDB .NET Core Preview SDK enables you to build fast, cross-platform [ASP.NET Core](#) and [.NET Core](#) apps to run on Windows, Mac, and Linux, as well as create [Universal Windows Platform \(UWP\)](#) apps.

The DocumentDB .NET Core Preview SDK has feature parity with the latest version of the [DocumentDB .NET SDK](#) and supports the following:

- All [connection modes](#): Gateway mode, Direct TCP, and Direct HTTPs.
- All [consistency levels](#): Strong, Session, Bounded Staleness, and Eventual.
- [Partitioned collections](#).
- [Multi-region database accounts and geo-replication](#).

If you have questions related to this SDK, post to [StackOverflow](#), the [MSDN forums](#), or send email to askdocdb@microsoft.com. Or file an issue in the [github repository](#).

Release & Retirement Dates

VERSION	RELEASE DATE	RETIREMENT DATE
0.1.0-preview	November 15, 2016	---

See Also

To learn more about DocumentDB, see [Microsoft Azure DocumentDB service page](#).

DocumentDB .NET examples

11/15/2016 • 4 min to read • [Edit on GitHub](#)

Contributors

[Rajesh Nagpal](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [mimig](#) • [Andrew Liu](#) • [jayantacs](#) • [Ryan CrawCour](#)

Sample solutions that perform CRUD operations and other common operations on Azure DocumentDB resources are included in the [azure-documentdb-net](#) GitHub repository. This article provides:

- Links to the tasks in each of the example C# project files.
- Links to the related API reference content.

Prerequisites

1. You need an Azure account to use these NoSQL examples:
 - You can [open an Azure account for free](#): You get credits you can use to try out paid Azure services, and even after they're used up you can keep the account and use free Azure services, such as Websites. Your credit card will never be charged, unless you explicitly change your settings and ask to be charged.
 - You can [activate Visual Studio subscriber benefits](#): Your Visual Studio subscription gives you credits every month that you can use for paid Azure services.
2. You also need the [Microsoft.Azure.DocumentDB NuGet package](#).

NOTE

Each sample is self-contained, it sets itself up and cleans up after itself. As such, the samples issue multiple calls to `CreateDocumentCollectionAsync()`. Each time this is done your subscription is billed for 1 hour of usage per the performance tier of the collection being created.

Database examples

The [RunDatabaseDemo](#) method of the sample of the DatabaseManagement project shows how to perform the following tasks.

TASK	API REFERENCE
Create a database	DocumentClient.CreateDatabaseAsync
Query an account for a database	DocumentQueryable.CreateDatabaseQuery
Read a database by Id	DocumentClient.ReadDatabaseAsync
List databases for an account	DocumentClient.ReadDatabaseFeedAsync
Delete a database	DocumentClient.DeleteDatabaseAsync

Collection examples

The [RunCollectionDemo](#) method of the sample CollectionManagement project shows how to do the following

tasks.

TASK	API REFERENCE
Create a collection	DocumentClient.CreateDocumentCollectionAsync
Get performance tier of a collection	DocumentQueryable.CreateOfferQuery
Change performance tier of a collection	DocumentClient.ReplaceOfferAsync
Get a collection by Id	DocumentClient.ReadDocumentCollectionAsync
Read a list of all collections in a database	DocumentClient.ReadDocumentCollectionFeedAsync
Delete a collection	DocumentClient.DeleteDocumentCollectionAsync

Document examples

The [RunDocumentsDemo](#) method of the sample DocumentManagement project shows how to do the following tasks.

TASK	API REFERENCE
Create a document	DocumentClient.CreateDocumentAsync
Read a document by Id	DocumentClient.ReadDocumentAsync
Read all documents in a collection	DocumentClient.ReadDocumentFeedAsync
Query for documents	DocumentClient.CreateDocumentQuery
Replace a document	DocumentClient.ReplaceDocumentAsync
Upsert a document	DocumentClient.UpsertDocumentAsync
Delete document	DocumentClient.DeleteDocumentAsync
Working with .NET dynamic objects	DocumentClient.CreateDocumentAsync DocumentClient.ReadDocumentAsync DocumentClient.ReplaceDocumentAsync
Replace document with conditional ETag check	DocumentClient.AccessCondition Documents.Client.AccessConditionType
Read document only if document has changed	DocumentClient.AccessCondition Documents.Client.AccessConditionType

Indexing examples

The [RunIndexDemo](#) method of the sample IndexManagement project shows how to perform the following tasks.

TASK	API REFERENCE
Exclude a document from the index	IndexingDirective.Exclude
Use manual (instead of automatic) indexing	IndexingPolicy.Automatic
Use lazy (instead of consistent) indexing	IndexingMode.Lazy
Exclude specified document paths from the index	IndexingPolicy.ExcludedPaths
Force a range scan operation on a hash indexed path	FeedOptions.EnableScanInQuery
Use range indexes on strings	IndexingPolicy.IncludedPaths RangeIndex
Perform an index transform	ReplaceDocumentCollectionAsync

For more information about indexing, see [DocumentDB indexing policies](#).

Partitioning examples

The partitioning sample file, [azure-documentdb-net/samples/code-samples/Partitioning/Program.cs](#), shows how to do the following tasks. In some cases, additional helper files are used to complete the task.

TASK	API REFERENCE
Use a HashPartitionResolver	HashPartitionResolver
Use a RangePartitionResolver	Range with RangePartitionResolver
Implement custom partition resolvers	IPartitionResolver
Implement a simple lookup table with LookupPartitionResolver.cs	RangePartitionResolver
Implement a partition resolver that creates or clones collections with ManagedHashPartitionResolver.cs	IPartitionResolver
Implement a spillover scheme with SpilloverPartitionResolver.cs	IPartitionResolver
Saving and loading resolver configs with RunSerializeDeserializeSample	IPartitionResolver
Add, remove, and re-balance data among partitions with RepartitionDataSample and DocumentClientHashPartitioningManager.cs	HashPartitionResolver
Implement a partition resolver for routing during repartitioning	IPartitionResolver

For more information about partitioning and sharding, see [Partition and scale data in DocumentDB](#).

Geospatial examples

The geospatial sample file, [azure-documentdb-net/samples/code-samples/Geospatial/Program.cs](https://github.com/Azure-Samples/azure-documentdb-net/samples/code-samples/Geospatial/Program.cs), shows how to do the following tasks.

TASK	API REFERENCE
Enable geospatial indexing on a new collection	IndexingPolicy IndexKind.Spatial DataType.Point
Insert documents with GeoJSON points	DocumentClient.CreateDocumentAsync DataType.Point
Find points within a specified distance	ST_DISTANCE or [GeometryOperationExtensions.Distance] (https://msdn.microsoft.com/library/azure/microsoft.azure.documents.spatial.geometryoperationextensions.distance.aspx#M:Microsoft.Azure.Documents.Spatial.GeometryOperationExtensions.Distance(Microsoft.Azure.Documents.Spatial.Geometry,Microsoft.Azure.Documents.Spatial.Geometry))
Find points within a polygon	ST_WITHIN or [GeometryOperationExtensions.Within] (https://msdn.microsoft.com/library/azure/microsoft.azure.documents.spatial.geometryoperationextensions.within.aspx#M:Microsoft.Azure.Documents.Spatial.GeometryOperationExtensions.Within(Microsoft.Azure.Documents.Spatial.Geometry,Microsoft.Azure.Documents.Spatial.Geometry)andPolygon) and Polygon
Enable geospatial indexing on an existing collection	DocumentClient.ReplaceDocumentCollectionAsync DocumentCollection.IndexingPolicy
Validate point and polygon data	ST_ISVALID ST_ISVALIDDETAILED GeometryOperationExtensions.IsValid GeometryOperationExtensions.IsValidDetailed

For more information about working with Geospatial data, see [Working with Geospatial data in Azure DocumentDB](#).

Query examples

The query document file, [azure-documentdb-net/samples/code-samples/Queries/Program.cs](https://github.com/Azure-Samples/azure-documentdb-net/samples/code-samples/Queries/Program.cs), shows how to do each of the following tasks using the SQL query grammar, the LINQ provider with query, and with Lambda.

TASK	API REFERENCE
Query for all documents	DocumentQueryable.CreateDocumentQuery
Query for equality using ==	DocumentQueryable.CreateDocumentQuery
Query for inequality using != and NOT	DocumentQueryable.CreateDocumentQuery
Query using range operators like >, <, >=, <=	DocumentQueryable.CreateDocumentQuery

TASK	API REFERENCE
Query using range operators against strings	DocumentQueryable.CreateDocumentQuery
Query with Order by	DocumentQueryable.CreateDocumentQuery
Work with subdocuments	DocumentQueryable.CreateDocumentQuery
Query with intra-document Joins	DocumentQueryable.CreateDocumentQuery
Query with string, math and array operators	DocumentQueryable.CreateDocumentQuery
Query with parameterized SQL using SqlQuerySpec	DocumentQueryable.CreateDocumentQuery SqlQuerySpec
Query with explicit paging	DocumentQueryable.CreateDocumentQuery
Query partitioned collections in parallel	DocumentQueryable.CreateDocumentQuery
Query with Order by for partitioned collections	DocumentQueryable.CreateDocumentQuery

For more information about writing queries, see [SQL query within DocumentDB](#).

Server-side programming examples

The server-side programming file, [azure-documentdb-net/samples/code-samples/ServerSideScripts/Program.cs](#), shows how to do the following tasks.

TASK	API REFERENCE
Create a stored procedure	DocumentClient.CreateStoredProcedureAsync
Execute a stored procedure	DocumentClient.ExecuteStoredProcedureAsync
Read a document feed for a stored procedure	DocumentClient.ReadDocumentFeedAsync
Create a stored procedure with Order by	DocumentClient.CreateStoredProcedureAsync
Create a pre-trigger	DocumentClient.CreateTriggerAsync
Create a post-trigger	DocumentClient.CreateTriggerAsync
Create a User Defined Function (UDF)	DocumentClient.CreateUserDefinedFunctionAsync

For more information about server-side programming, see [DocumentDB server-side programming: Stored procedures, database triggers, and UDFs](#).

User management examples

The user management file, [azure-documentdb-net/samples/code-samples/UserManagement/Program.cs](#), shows how to do the following tasks.

TASK	API REFERENCE
Create a user	DocumentClient.CreateUserAsync
Set permissions on a collection or document	DocumentClient.CreatePermissionAsync
Get a list of a user's permissions	DocumentClient.ReadUserAsync DocumentClient.ReadPermissionFeedAsync

DocumentDB APIs and SDKs

11/22/2016 • 5 min to read • [Edit on GitHub](#)

Contributors

Rajesh Nagpal • [mimig](#) • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [Mo Derakhshani](#) • [Andrew Liu](#) • [PRmerger](#)
• [Jason Card](#) • [Ryan CrawCour](#) • [Ross McAllister](#)

DocumentDB Node.js API and SDK

Download SDK	NPM
API documentation	Node.js API reference documentation
SDK installation instructions	Installation instructions
Contribute to SDK	GitHub
Samples	Node.js code samples
Get started tutorial	Get started with the Node.js SDK
Web app tutorial	Build a Node.js web application using DocumentDB
Current supported platform	Node.js v0.10 Node.js v0.12 Node.js v4.2.0

Release notes

1.10.0

- Added support for cross partition parallel queries.
- Added support for TOP/ORDER BY queries for partitioned collections.

1.9.0

- Added retry policy support for throttled requests. (Throttled requests receive a request rate too large exception, error code 429.) By default, DocumentDB retries nine times for each request when error code 429 is encountered, honoring the retryAfter time in the response header. A fixed retry interval time can now be set as part of the RetryOptions property on the ConnectionPolicy object if you want to ignore the retryAfter time returned by server between the retries. DocumentDB now waits for a maximum of 30 seconds for each request that is being throttled (irrespective of retry count) and returns the response with error code 429. This time can also be overridden in the RetryOptions property on ConnectionPolicy object.
- DocumentDB now returns x-ms-throttle-retry-count and x-ms-throttle-retry-wait-time-ms as the response headers in every request to denote the throttle retry count and the cumulative time the request waited between the retries.
- The RetryOptions class was added, exposing the RetryOptions property on the ConnectionPolicy class that can be used to override some of the default retry options.

1.8.0

- Added the support for multi-region database accounts.

1.7.0

- Added the support for Time To Live(TTL) feature for documents.

1.6.0

- Implemented [partitioned collections](#) and [user-defined performance levels](#).

1.5.6

- Fixed RangePartitionResolver.resolveForRead bug where it was not returning links due to a bad concat of results.

1.5.5

- Fixed hashPartitionResolver resolveForRead(): When no partition key supplied was throwing exception, instead of returning a list of all registered links.

1.5.4

- Fixes issue [#100](#) - Dedicated HTTPS Agent: Avoid modifying the global agent for DocumentDB purposes. Use a dedicated agent for all of the lib's requests.

1.5.3

- Fixes issue [#81](#) - Properly handle dashes in media ids.

1.5.2

- Fixes issue [#95](#) - EventEmitter listener leak warning.

1.5.1

- Fixes issue [#92](#) - rename folder Hash to hash for case sensitive systems.

1.5.0

- Implement sharding support by adding hash & range partition resolvers.

1.4.0

- Implement Upsert. New upsertXXX methods on documentClient.

1.3.0

- Skipped to bring version numbers in alignment with other SDKs.

1.2.2

- Split Q promises wrapper to new repository.
- Update to package file for npm registry.

1.2.1

- Implements ID Based Routing.
- Fixes Issue [#49](#) - current property conflicts with method current().

1.2.0

- Added support for GeoSpatial index.
- Validates id property for all resources. Ids for resources cannot contain ?, /, #, //, characters or end with a space.
- Adds new header "index transformation progress" to ResourceResponse.

1.1.0

- Implements V2 indexing policy.

1.0.3

- Issue [#40](#) - Implemented eslint and grunt configurations in the core and promise SDK.

1.0.2

- Issue [#45](#) - Promises wrapper does not include header with error.

1.0.1

- Implemented ability to query for conflicts by adding readConflicts, readConflictAsync, and queryConflicts.
- Updated API documentation.
- Issue [#41](#) - client.createDocumentAsync error.

1.0.0

- GA SDK.

Release & Retirement Dates

Microsoft will provide notification at least **12 months** in advance of retiring an SDK in order to smooth the transition to a newer/supported version.

New features and functionality and optimizations are only added to the current SDK, as such it is recommend that you always upgrade to the latest SDK version as early as possible.

Any request to DocumentDB using a retired SDK will be rejected by the service.

WARNING

All versions of the Azure DocumentDB SDK for Node.js prior to version **1.0.0** will be retired on **February 29, 2016**.

VERSION	RELEASE DATE	RETIREMENT DATE
1.10.0	October 03, 2016	---
1.9.0	July 07, 2016	---
1.8.0	June 14, 2016	---
1.7.0	April 26, 2016	---
1.6.0	March 29, 2016	---
1.5.6	March 08, 2016	---
1.5.5	February 02, 2016	---
1.5.4	February 01, 2016	---
1.5.2	January 26, 2016	---
1.5.2	January 22, 2016	---
1.5.1	January 4, 2016	---

VERSION	RELEASE DATE	RETIREMENT DATE
1.5.0	December 31, 2015	---
1.4.0	October 06, 2015	---
1.3.0	October 06, 2015	---
1.2.2	September 10, 2015	---
1.2.1	August 15, 2015	---
1.2.0	August 05, 2015	---
1.1.0	July 09, 2015	---
1.0.3	June 04, 2015	---
1.0.2	May 23, 2015	---
1.0.1	May 15, 2015	---
1.0.0	April 08, 2015	---
0.9.4-prerelease	April 06, 2015	February 29, 2016
0.9.3-prerelease	January 14, 2015	February 29, 2016
0.9.2-prerelease	December 18, 2014	February 29, 2016
0.9.1-prerelease	August 22, 2014	February 29, 2016
0.9.0-prerelease	August 21, 2014	February 29, 2016

FAQ

1. How will customers be notified of the retiring SDK?

Microsoft will provide 12 month advance notification to the end of support of the retiring SDK in order to facilitate a smooth transition to a supported SDK. Further, customers will be notified through various communication channels – Azure Management Portal, Developer Center, blog post, and direct communication to assigned service administrators.

2. Can customers author applications using a "to-be" retired DocumentDB SDK during the 12 month period?

Yes, customers will have full access to author, deploy and modify applications using the "to-be" retired DocumentDB SDK during the 12 month grace period. During the 12 month grace period, customers are advised to migrate to a newer supported version of DocumentDB SDK as appropriate.

3. Can customers author and modify applications using a retired DocumentDB SDK after the 12 month notification period?

After the 12 month notification period, the SDK will be retired. Any access to DocumentDB by an applications

using a retired SDK will not be permitted by the DocumentDB platform. Further, Microsoft will not provide customer support on the retired SDK.

4. What happens to Customer's running applications that are using unsupported DocumentDB SDK version?

Any attempts made to connect to the DocumentDB service with a retired SDK version will be rejected.

5. Will new features and functionality be applied to all non-retired SDKs

New features and functionality will only be added to new versions. If you are using an old, non-retired, version of the SDK your requests to DocumentDB will still function as previous but you will not have access to any new capabilities.

6. What should I do if I cannot update my application before a cut-off date

We recommend that you upgrade to the latest SDK as early as possible. Once an SDK has been tagged for retirement you will have 12 months to update your application. If, for whatever reason, you cannot complete your application update within this timeframe then please contact the [DocumentDB Team](#) and request their assistance before the cutoff date.

See also

To learn more about DocumentDB, see [Microsoft Azure DocumentDB](#) service page.

DocumentDB Node.js examples

11/15/2016 • 2 min to read • [Edit on GitHub](#)

Contributors

Mo Derakhshani • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • mimig • PRmerger • Andrew Liu • Ryan CrawCour
• Ross McAllister

Sample solutions that perform CRUD operations and other common operations on Azure DocumentDB resources are included in the [azure-documentdb-nodejs](#) GitHub repository. This article provides:

- Links to the tasks in each of the Node.js example project files.
- Links to the related API reference content.

Prerequisites

1. You need an Azure account to use these Node.js examples:
 - You can [open an Azure account for free](#): You get credits you can use to try out paid Azure services, and even after they're used up you can keep the account and use free Azure services, such as Websites. Your credit card will never be charged, unless you explicitly change your settings and ask to be charged.
 - You can [activate Visual Studio subscriber benefits](#): Your Visual Studio subscription gives you credits every month that you can use for paid Azure services.
2. You also need the [Node.js SDK](#).

NOTE

Each sample is self-contained, it sets itself up and cleans up after itself. As such, the samples issue multiple calls to [DocumentClient.createCollection](#). Each time this is done your subscription will be billed for 1 hour of usage per the performance tier of the collection being created.

Database examples

The [app.js](#) file of the [DatabaseManagement](#) project shows how to perform the following tasks.

TASK	API REFERENCE
Create a database	DocumentClient.createDatabase
Query an account for a database	DocumentClient.queryDatabases
Read a database by Id	DocumentClient.readDatabase
List databases for an account	DocumentClient.readDatabases
Delete a database	DocumentClient.deleteDatabase

Collection examples

The [app.js](#) file of the [CollectionManagement](#) project shows how to perform the following tasks.

TASK	API REFERENCE
Create a collection	DocumentClient.createCollection
Read a list of all collections in a database	DocumentClient.readCollections
Get a collection by <code>_self</code>	DocumentClient.readCollection
Get a collection by <code>Id</code>	DocumentClient.readCollection
Get performance tier of a collection	DocumentQueryable.queryOffers
Change performance tier of a collection	DocumentClient.replaceOffer
Delete a collection	DocumentClient.deleteCollection

Document examples

The [app.js](#) file of the [DocumentManagement](#) project shows how to perform the following tasks.

TASK	API REFERENCE
Create documents	DocumentClient.createDocument
Read the document feed for a collection	DocumentClient.readDocument
Read a document by ID	DocumentClient.readDocument
Read document only if document has changed	DocumentClient.readDocument RequestOptions.accessCondition
Query for documents	DocumentClient.queryDocuments
Replace a document	DocumentClient.replaceDocument
Replace document with conditional ETag check	DocumentClient.replaceDocument RequestOptions.accessCondition
Delete a document	DocumentClient.deleteDocument

Indexing examples

The [app.js](#) file of the [IndexManagement](#) project shows how to perform the following tasks.

TASK	API REFERENCE
Create a collection with default indexing	DocumentClient.createDocument
Manually index a specific document	indexingDirective: 'include'
Manually exclude a specific document from the index	RequestOptions.indexingDirective

TASK	API REFERENCE
Use lazy indexing for bulk import or read heavy collections	IndexingMode.Lazy
Include specific paths of a document in indexing	IndexingPolicy.IncludedPaths
Exclude certain paths from indexing	ExcludedPath
Allow a scan on a string path during a range operation	ExcludedPath.EnableScanInQuery
Create a range index on a string path	DocumentClient.queryDocument
Create a collection with default indexPolicy, then update this online	DocumentClient.createCollection DocumentClient.replaceCollection#replaceCollection

For more information about indexing, see [DocumentDB indexing policies](#).

Server-side programming examples

The [app.js](#) file of the [ServerSideScripts](#) project shows how to perform the following tasks.

TASK	API REFERENCE
Create a stored procedure	DocumentClient.createStoredProcedure
Execute a stored procedure	DocumentClient.executeStoredProcedure

For more information about server-side programming, see [DocumentDB server-side programming: Stored procedures, database triggers, and UDFs](#).

Partitioning examples

The [app.js](#) file of the [Partitioning](#) project shows how to perform the following tasks.

TASK	API REFERENCE
Use a HashPartitionResolver	HashPartitionResolver

For more information about partitioning data in DocumentDB, see [Partition and scale data in DocumentDB](#).

DocumentDB APIs and SDKs

11/22/2016 • 4 min to read • [Edit on GitHub](#)

Contributors

Rajesh Nagpal • mimig • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • PRmerger • Andrew Liu • Ross McAllister
• Ryan CrawCour • Andy Pasic

DocumentDB Python API and SDK

Download SDK	PyPI
API documentation	Python API reference documentation
SDK installation instructions	Python SDK installation instructions
Contribute to SDK	GitHub
Get started	Get started with the Python SDK
Current supported platform	Python 2.7 and Python 3.5

Release notes

2.0.1

- Made editorial changes to documentation comments.

2.0.0

- Added support for Python 3.5.
- Added support for connection pooling using a requests module.
- Added support for session consistency.
- Added support for TOP/ORDERBY queries for partitioned collections.

1.9.0

- Added retry policy support for throttled requests. (Throttled requests receive a request rate too large exception, error code 429.) By default, DocumentDB retries nine times for each request when error code 429 is encountered, honoring the retryAfter time in the response header. A fixed retry interval time can now be set as part of the RetryOptions property on the ConnectionPolicy object if you want to ignore the retryAfter time returned by server between the retries. DocumentDB now waits for a maximum of 30 seconds for each request that is being throttled (irrespective of retry count) and returns the response with error code 429. This time can also be overridden in the RetryOptions property on ConnectionPolicy object.
- DocumentDB now returns x-ms-throttle-retry-count and x-ms-throttle-retry-wait-time-ms as the response headers in every request to denote the throttle retry count and the cumulative time the request waited between the retries.
- Removed the RetryPolicy class and the corresponding property (retry_policy) exposed on the document_client class and instead introduced a RetryOptions class exposing the RetryOptions property on ConnectionPolicy

class that can be used to override some of the default retry options.

1.8.0

- Added the support for multi-region database accounts.

1.7.0

- Added the support for Time To Live(TTL) feature for documents.

1.6.1

- Bug fixes related to server side partitioning to allow special characters in partitionkey path.

1.6.0

- Implemented [partitioned collections](#) and [user-defined performance levels](#).

1.5.0

- Add Hash & Range partition resolvers to assist with sharding applications across multiple partitions.

1.4.2

- Implement Upsert. New UpsertXXX methods added to support Upsert feature.
- Implement ID Based Routing. No public API changes, all changes internal.

1.2.0

- Supports GeoSpatial index.
- Validates id property for all resources. Ids for resources cannot contain ?, /, #, \ characters or end with a space.
- Adds new header "index transformation progress" to ResourceResponse.

1.1.0

- Implements V2 indexing policy.

1.0.1

- Supports proxy connection.

1.0.0

- GA SDK.

Release & retirement dates

Microsoft will provide notification at least **12 months** in advance of retiring an SDK in order to smooth the transition to a newer/supported version.

New features and functionality and optimizations are only added to the current SDK, as such it is recommend that you always upgrade to the latest SDK version as early as possible.

Any request to DocumentDB using a retired SDK will be rejected by the service.

WARNING

All versions of the Azure DocumentDB SDK for Python prior to version 1.0.0 will be retired on **February 29, 2016**.

VERSION	RELEASE DATE	RETIREMENT DATE
2.0.1	October 30, 2016	---

VERSION	RELEASE DATE	RETIREMENT DATE
2.0.0	September 29, 2016	---
1.9.0	July 07, 2016	---
1.8.0	June 14, 2016	---
1.7.0	April 26, 2016	---
1.6.1	April 08, 2016	---
1.6.0	March 29, 2016	---
1.5.0	January 03, 2016	---
1.4.2	October 06, 2015	---
1.4.1	October 06, 2015	---
1.2.0	August 06, 2015	---
1.1.0	July 09, 2015	---
1.0.1	May 25, 2015	---
1.0.0	April 07, 2015	---
0.9.4-prelease	January 14, 2015	February 29, 2016
0.9.3-prelease	December 09, 2014	February 29, 2016
0.9.2-prelease	November 25, 2014	February 29, 2016
0.9.1-prelease	September 23, 2014	February 29, 2016
0.9.0-prelease	August 21, 2014	February 29, 2016

FAQ

1. How will customers be notified of the retiring SDK?

Microsoft will provide 12 month advance notification to the end of support of the retiring SDK in order to facilitate a smooth transition to a supported SDK. Further, customers will be notified through various communication channels – Azure Management Portal, Developer Center, blog post, and direct communication to assigned service administrators.

2. Can customers author applications using a "to-be" retired DocumentDB SDK during the 12 month period?

Yes, customers will have full access to author, deploy and modify applications using the "to-be" retired DocumentDB SDK during the 12 month grace period. During the 12 month grace period, customers are advised to migrate to a newer supported version of DocumentDB SDK as appropriate.

3. Can customers author and modify applications using a retired DocumentDB SDK after the 12 month notification period?

After the 12 month notification period, the SDK will be retired. Any access to DocumentDB by an applications using a retired SDK will not be permitted by the DocumentDB platform. Further, Microsoft will not provide customer support on the retired SDK.

4. What happens to Customer's running applications that are using unsupported DocumentDB SDK version?

Any attempts made to connect to the DocumentDB service with a retired SDK version will be rejected.

5. Will new features and functionality be applied to all non-retired SDKs

New features and functionality will only be added to new versions. If you are using an old, non-retired, version of the SDK your requests to DocumentDB will still function as previous but you will not have access to any new capabilities.

6. What should I do if I cannot update my application before a cut-off date

We recommend that you upgrade to the latest SDK as early as possible. Once an SDK has been tagged for retirement you will have 12 months to update your application. If, for whatever reason, you cannot complete your application update within this timeframe then please contact the [DocumentDB Team](#) and request their assistance before the cutoff date.

See also

To learn more about DocumentDB, see [Microsoft Azure DocumentDB](#) service page.

DocumentDB Python examples

11/15/2016 • 1 min to read • [Edit on GitHub](#)

Contributors

[Mo Derakhshani](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [mimig](#) • [Andrew Liu](#) • [Ryan CrawCour](#)

Sample solutions that perform CRUD operations and other common operations on Azure DocumentDB resources are included in the [azure-documentdb-python](#) GitHub repository. This article provides:

- Links to the tasks in each of the Python example project files.
- Links to the related API reference content.

Prerequisites

1. You need an Azure account to use these Python examples:
 - You can [open an Azure account for free](#): You get credits you can use to try out paid Azure services, and even after they're used up you can keep the account and use free Azure services, such as Websites. Your credit card will never be charged, unless you explicitly change your settings and ask to be charged.
 - You can [activate Visual Studio subscriber benefits](#): Your Visual Studio subscription gives you credits every month that you can use for paid Azure services.
2. You also need the [Python SDK](#).

NOTE

Each sample is self-contained, it sets itself up and cleans up after itself. As such, the samples issue multiple calls to `document_client.CreateCollection`. Each time this is done your subscription will be billed for 1 hour of usage per the performance tier of the collection being created.

Database examples

The [Program.py](#) file of the [DatabaseManagement](#) project shows how to perform the following tasks.

TASK	API REFERENCE
Create a database	<code>document_client.CreateDatabase</code>
Query an account for a database	<code>document_client.QueryDatabases</code>
Read a database by Id	<code>document_client.ReadDatabase</code>
List databases for an account	<code>document_client.ReadDatabases</code>
Delete a database	<code>document_client.DeleteDatabase</code>

Collection examples

The [Program.py](#) file of the [CollectionManagement](#) project shows how to perform the following tasks.

TASK	API REFERENCE
Create a collection	document_client.CreateCollection
Read a list of all collections in a database	document_client.ListCollections
Get a collection by Id	document_client.ReadCollection
Get performance tier of a collection	DocumentQueryable.QueryOffers
Change performance tier of a collection	document_client.ReplaceOffer
Delete a collection	document_client.DeleteCollection

DocumentDB SQL query cheat sheet PDF

11/15/2016 • 1 min to read • [Edit on GitHub](#)

Contributors

[mimig](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#)

The **DocumentDB SQL Query Cheat Sheet** helps you quickly write queries for DocumentDB data by displaying common database queries, keywords, built-in functions, and operators in an easy to print PDF reference sheet.

DocumentDB supports relational, hierarchical, and spatial querying of JSON documents using [SQL](#) without specifying a schema or secondary indexes. In addition to the standard ANSI-SQL keywords and operators, DocumentDB supports JavaScript user defined functions (UDFs), JavaScript operators, and a multitude of built-in functions.

Download the DocumentDB SQL query cheat sheet PDF

Write your queries faster by downloading the SQL query cheat sheet and using it as a quick reference. The SQL cheat sheet PDF shows common queries used to retrieve data from two example JSON documents. To keep it nearby, you can print the single-sided SQL query cheat sheet in page letter size (8.5 x 11 in.).

Download the SQL cheat sheet here: [Microsoft Azure DocumentDB SQL cheat sheet](#)



Microsoft Azure DocumentDB Query Cheat Sheet

This cheat sheet helps you quickly write DocumentDB queries by showing some common SQL queries used to retrieve data from two simple JSON documents.

Example Family JSON Documents		SQL Query	Sample Queries
<pre>{ "id": "AndersonFamily", "lastName": "Anderson", "parents": [{ "firstName": "Thomas", "lastName": "Mary Kay" }], "children": [{ "firstName": "Marguerite Thaulov", "gender": "Female", "grade": 8, "pets": [{ "givenName": "Fluffy" }] }, { "state": "WA", "county": "King", "city": "Seattle", "createdAtUtc": "2015-01-01T12:00Z", "isRegistered": true, "location": { "type": "Null", "coordinates": { "lat": 4.0 } } }] }</pre>	1	<pre>-- Find families by ID SELECT * FROM Families f WHERE f.id = "AndersonFamily" -- [[{ "id": "AndersonFamily", "lastName": "Anderson", "parents": [{ "firstName": "Thomas", "lastName": "Mary Kay" }], "children": [{ "firstName": "Marguerite Thaulov", "gender": "Female", "grade": 8, "pets": [{ "givenName": "Fluffy" }] }, { "state": "WA", "county": "King", "city": "Seattle", "createdAtUtc": "2015-01-01T12:00Z", "isRegistered": true, "location": { "type": "Null", "coordinates": { "lat": 4.0 } } }] }]</pre>	Comparison operators <pre>SELECT * FROM Families children(c) c WHERE c.grade > 8</pre> Logical operators <pre>SELECT * FROM Families children(c) c WHERE c.grade > 8 AND c.isRegistered = true</pre> ORDER BY keyword <pre>SELECT f.id, f.address.city FROM Families f ORDER BY f.address.city</pre> IN keyword <pre>SELECT * FROM Families WHERE Families.address.state IN ("WA", "AZ", "CA", "PA", "OH", "OR", "NY", "UT")</pre> Ternary (?) and Coalesce (?) operators <pre>SELECT (c.grade < 8) ? "Elementary" : (c.grade < 8) ? "Junior" : "High" AS gradeLevel FROM Families children(c) c</pre> Escape/quoted characters <pre>SELECT f["lastName"] FROM Families f WHERE f["id"] = "AndersonFamily"</pre> Object/Array Creation <pre>SELECT [f.address.city, f.address.state] AS CityState FROM Families f</pre> Value keyword <pre>SELECT VALUE "Hello World"</pre>
<pre>{ "id": "AndersonFamily", "parents": [{ "familyName": "Anderson", "givenName": "Jesse" }, { "familyName": "Miller", "givenName": "Lisa" }], "children": [{ "familyName": "Anderson", "givenName": "Jesse", "gender": "Female", "grade": 8, "pets": [{ "givenName": "Tooty" }] }, { "familyName": "Sheldon" }] }</pre>	2	<pre>-- Register UDF for HASK_JOIN with this code function (input, pattern) { return input.match(pattern) != null; } -- Use the UDF SELECT udf_HASK_JOIN(families.address.city, ".*Seattle") [[{ "id": true }, { "id": false }]</pre>	SQL + JavaScript UDF <pre>-- Register UDF for HASK_JOIN with this code function (input, pattern) { return input.match(pattern) != null; } -- Use the UDF SELECT udf_HASK_JOIN(families.address.city, ".*Seattle")</pre> Intra-document JOINs <pre>SELECT * FROM Families f WHERE f.lastName = @lastName AND f.address.state = @addressState</pre> Parameterized SQL <pre>SELECT Families.id, Families.address.city FROM Families WHERE STARTswith(Families.id, "Anderson")</pre> String Built-in functions <pre>SELECT Families.id FROM Families WHERE array_contains(families.parents, { givenName: "Kobin", familyName: "Anderson" })</pre> Array Built-in functions <pre>SELECT VALUE ABS(-4)</pre> Math Built-in functions <pre>SELECT IS_DEFINED(f.lastName), IS_NULL(ABS(-4)) FROM Families f</pre> Type Built-in functions <pre>SELECT * FROM Families children(c) c WHERE c.grade BETWEEN 1 AND 8</pre> BETWEEN keyword <pre>SELECT TOP 100 * FROM Families f</pre> TOP keyword <pre>SELECT * FROM Families f WHERE AT_DISTANCE(f.location, {"type": "Point", "coordinates": [11.9, -4.5]}) < 10000</pre> Geospatial functions
Built-in functions		Operators	
Mathematical	ABS, CEILING, EXP, FLOOR, LOG, LOG10, POWER, ROUND, SIGN, SQRT, SQUARE, TRUNC, ABS, ACOS, ASIN, ATAN, COS, COS, DEGREES, PI, RADIANS, SIN, AND TAN	Arithmetic	+, -, *, /, %
		Bitwise	~, &, ^, <<., >> (zero-fill right shift)
Type checking	IS_ARRAY, IS_NULL, IS_NULL, IS_NUMBER, IS_OBJECT, IS_STRING, IS_BOOLEAN, AND IS_BOOLEAN	Logical	AND, OR, NOT
		Comparison	=, !=, >, >=, <, <=, <>, ??
String	CONCAT, CONCAT_WS, SUBSTRING, SUBSTR, LEFT, LENGTH, LOWER, LTRIM, REPLACE, REVERSE, RTRIM, STRLEN, SUBSTRING, SUBSTRING2, AND UPPER	String	(concatenate)
		Ternary	?
Array		Query Interfaces	
Geospatial	ST_DISTANCE, ST_DISTANCE, ST_INTERSECTS, ST_OVERLAP, AND ST_OVERLAP	Server-side	SQL, JavaScript Integrated query
		Client-side	.NET (LINQ), Java, JavaScript, Node.js, Python

www.documentsdb.com | Online Query Playground: www.documentsdb.com/sql/demo | RU Estimator www.documentsdb.com/capacityplanner
Tweet: @documentsdb | StackOverflow: #azure-database

www.documentdb.com | Online Query Playground: www.documentdb.com/sql/demo | RU Estimator: www.documentdb.com/capacityplanner
Tweet: @documentdb | StackOverflow: #azure-documentdb

More help with writing SQL queries

- For a walk through of the query options available in DocumentDB, see [Query DocumentDB](#).
- For the related reference documentation, see [DocumentDB SQL Query Language](#).

Release notes

Updated on 7/29/2016 to include TOP.

Community portal

11/22/2016 • 9 min to read • [Edit on GitHub](#)

Contributors

[Andrew Liu](#) • [mimig](#) • [Andy Pasic](#) • [Kim Whitlatch \(Beyondsoft Corporation\)](#) • [Tyson Nevil](#) • [Ross McAllister](#)

Community spotlight

Let us promote your project! Show us the awesome project you're working on with DocumentDB, and we will help share your genius with the world. To submit your project, send us an e-mail at: askdocdb@microsoft.com.

documentdb-lumenize

by Larry Maccherone

Aggregations (Group-by, Pivot-table, and N-dimensional Cube) and Time Series Transformations as Stored Procedures in DocumentDB.

Check it out on [Github](#) and [npm](#).

DocumentDB Studio

by Ming Liu

A client management viewer/explorer for Microsoft Azure DocumentDB service.

Check it out on [Github](#).

DoQmentDB

by Ariel Mashraki

DoQmentDB is a Node.js promise-based client, that provides a MongoDB-like layer on top of DocumentDB.

Check it out on [Github](#) and [npm](#).

Swagger REST API for DocumentDB

by Howard Edidin

A DocumentDB REST API Swagger file that can be easily deployed as an API App.

Check it out on [Github](#).

fluent-plugin-documentdb

by Yoichi Kawasaki

fluent-plugin-documentdb is a Fluentd plugin for outputting to Azure DocumentDB.

Check it out on [Github](#) and [rubygems](#).

Find more open source DocumentDB projects on [GitHub](#).

News, blogs, and articles

You can stay up-to-date with the latest DocumentDB news and features by following [our blog](#).

Community posts:

- [A Journey to Social](#) - by *Matías Quaranta*
- [Azure DocumentDB protocol support for MongoDB in Preview, my test with Sitecore](#) - by *Mathieu Benoit*
- [Going Social with DocumentDB](#) - by *Matías Quaranta*
- [UWP, Azure App Services, and DocumentDB Soup: A photo-sharing app](#) - by *Eric Langland*
- [Notifications for new or changed DocumentDB resources using Logic Apps](#) - by *Howard Edidin*
- [Collecting logs in to Azure DocumentDB using fluent-plugin-documentdb](#) - by *Yoichi Kawasaki*
- [DocumentDB revisited Part 1/2 – The theory](#) - by *Peter Mannerhult*
- [What to love and hate about Azure’s DocumentDB](#) - by *George Saadeh*
- [Azure DocumentDB Server-Side Scripting](#) - by *Robert Sheldon*
- [DocumentDB as a data sink for Azure Stream Analytics](#) - by *Jan Hentschel*
- [Azure DocumentDB in production!](#) - by *Alexandre Walsh and Marc-Olivier Duval*
- [Azure Search Indexers – DocumentDB Queries \(Spanish\)](#) - by *Matthias Quaranta*
- [Azure DocumentDB SQL query basics \(Japanese\)](#) - by *Atsushi Yokohama*
- [Data Points - Aurelia Meets DocumentDB: A Matchmaker’s Journey](#) - by *Julie Lerman*
- [Infrastructure as Code and Continuous Deployment of a Node.js + Azure DocumentDB Solution](#) - by *Thiago Almedia*
- [Why DocumentDb Makes Good Business Sense for Some Projects](#) - by *Samuel Uresin*
- [Azure DocumentDB development moving forward – development of the Client class \(1 of 2\) \(Japanese\)](#) - by *Atsushi Yokohama*
- [Things you need to know when using Azure DocumentDB \(Japanese\)](#) - by *Atsushi Yokohama*
- [Dealing with RequestRateTooLarge errors in Azure DocumentDB and testing performance](#) - by *Azim Uddin*
- [Data Points - An Overview of Microsoft Azure DocumentDB](#) - by *Julie Lerman*
- [Using DocumentDB With F#](#) - by *Jamie Dixon*
- [Analysing Application Logs with DocumentDB](#) - by *Vincent-Philippe Lauzon*
- [Azure DocumentDB – Point in time Backups](#) - by *Juan Carlos Sanchez*

Do you have a blog post, code sample, or case-study you'd like to share? [Let us know!](#)

Events and recordings

Recent and upcoming events

EVENT NAME	SPEAKER	LOCATION	DATE	HASHTAG
Wintellect webinar: An Introduction to Azure DocumentDB	Josh Lane	Online	December 15, 2016 1pm EST	n/a

Are you speaking at or hosting an event? [Let us know](#) how we can help!

Previous events and recordings

EVENT NAME	SPEAKER	LOCATION	DATE	RECORDING
Connect(); // 2016	Kirill Gavrylyuk	New York, NY	November 16-18, 2016	Channel 9 Connect(); videos
Capital City .NET Users Group	Santosh Hari	Tallahassee, FL	November 3, 2016	n/a

EVENT NAME	SPEAKER	LOCATION	DATE	RECORDING
Ignite 2016	DocumentDB team	Atlanta, GA	September 26-30, 2016	Slidedeck
DevTeach	Ken Cenerelli	Montreal, Canada	July 4-8, 2016	NoSQL, No Problem, Using Azure DocumentDB
Integration and IoT	Eldert Grootenboer	Kontich, Belgium	June 30, 2016	n/a
MongoDB World 2016	Kirill Gavrylyuk	New York, New York	June 28-29, 2016	n/a
Integration User Group	Howard S. Edidin	Webcast	June 20, 2016	Do Logic Apps support error handling?
Meetup: UK Azure User Group	Andrew Liu	London, UK	May 12, 2016	n/a
Meetup: ONETUG - Orlando .NET User Group	Santosh Hari	Orlando, FL	May 12, 2016	n/a
SQLBits XV	Andrew Liu, Aravind Ramachandran	Liverpool, UK	May 4-7, 2016	n/a
Meetup: NYC .NET Developers Group	Leonard Lobel	New York City, NY	April 21, 2016	n/a
Integration User Group	Howard Edidin	Webinar	April 25, 2016	n/a
Global Azure Bootcamp: SoCal	Leonard Lobel	Orange, CA	April 16, 2016	n/a
Global Azure Bootcamp: Redmond	David Makogon	Redmond, WA	April 16, 2016	n/a
SQL Saturday #481 - Israel 2016	Leonard Lobel	HaMerkaz, Israel	April 04, 2016	n/a
Build 2016	John Macintyre	San Francisco, CA	March 31, 2016	Delivering Applications at Scale with DocumentDB, Azure's NoSQL Document Database
SQL Saturday #505 - Belgium 2016	Mihail Mateev	Antwerp, Belgium	March 19, 2016	n/a
Meetup: CloudTalk	Kirat Pandya	Bellevue, WA	March 3, 2016	n/a
Meetup: Azure Austin	Merwan Chinta	Austin, TX	January 28, 2016	n/a

EVENT NAME	SPEAKER	LOCATION	DATE	RECORDING
Meetup: msdevmtl	Vincent-Philippe Lauzon	Montreal, QC, Canada	December 1, 2015	n/a
Meetup: SeattleJS	David Makogon	Seattle, WA	November 12, 2015	n/a
PASS Summit 2015	Jeff Renz, Andrew Hoh, Aravind Ramachandran, John Macintyre	Seattle, WA	October 27-30, 2015	Developing Modern Applications on Azure
CloudDevelop 2015	David Makogon, Ryan Crawcour	Columbus, OH	October 23, 2015	n/a
SQL Saturday #454 - Turin 2015	Marco De Nittis	Turin, Italy	October 10, 2015	n/a
SQL Saturday #430 - Sofia 2015	Leonard Lobel	Sofia, Bulgaria	October 10, 2015	n/a
SQL Saturday #444 - Kansas City 2015	Jeff Renz	Kansas City, MO	October 3, 2015	n/a
SQL Saturday #429 - Oporto 2015	Leonard Lobel	Oporto, Portugal	October 3, 2015	n/a
AzureCon	David Makogon, Ryan Crawcour, John Macintyre	Virtual Event	September 29, 2015	Azure data and analytics platform Working with NoSQL Data in DocumentDB
SQL Saturday #434 - Holland 2015	Leonard Lobel	Utrecht, Netherlands	September 26, 2015	Introduction to Azure DocumentDB
SQL Saturday #441 - Denver 2015	Jeff Renz	Denver, CO	September 19, 2015	n/a
Meetup: San Francisco Bay Area Azure Developers	Andrew Liu	San Francisco, CA	September 15, 2015	n/a
Belarus Azure User Group Meet-Up	Alex Zyl	Minsk, Belarus	September 9, 2015	Introduction to DocumentDB concept overview, consistency levels, sharding strategies
NoSQL Now!	David Makogon, Ryan Crawcour	San Jose, CA	August 18-20, 2015	n/a
@Scale Seattle	Dharma Shukla	Seattle, WA	June 17, 2015	Schema Agnostic Indexing with Azure DocumentDB
Tech Refresh 2015	Bruno Lopes	Lisbon, Portugal	June 15, 2015	DocumentDB 101

EVENT NAME	SPEAKER	LOCATION	DATE	RECORDING
SQL Saturday #417 - Sri Lanka 2015	Mihail Mateev	Colombo, Sri Lanka	June 06, 2015	n/a
Meetup: Seattle Scalability Meetup	Dharma Shukla	Seattle, WA	May 27, 2015	n/a
SQL Saturday #377 - Kiev 2015	Mihail Mateev	Kiev, Ukraine	May 23, 2015	n/a
Database Month	Dharma Shukla	New York, NY	May 19, 2015	Azure DocumentDB: Massively-Scalable,- Multi-Tenant Document Database Service
Meetup: London SQL Server User Group	Allan Mitchell	London, UK	May 19, 2015	n/a
DevIntersection	Andrew Liu	Scottsdale, AZ	May 18-21, 2015	n/a
Meetup: Seattle Web App Developers Group	Andrew Liu	Seattle, WA	May 14, 2015	n/a
Ignite	Andrew Hoh, John Macintyre	Chicago, IL	May 4-8, 2015	SELECT Latest FROM DocumentDB video DocumentDB and Azure HDInsight: Better Together video
Build 2015	Ryan Crawcour	San Francisco, CA	April 29 - May 1, 2015	Build the Next Big Thing with Azure's NoSQL Service: DocumentDB
Global Azure Bootcamp 2015 - Spain	Luis Ruiz Pavon, Roberto Gonzalez	Madrid, Spain	April 25, 2015	#DEAN DocumentDB + Express + AngularJS + NodeJS running on Azure
Meetup: Azure Usergroup Denmark	Christian Holm Diget	Copenhagen, Denmark	April 16, 2015	n/a
Meetup: Charlotte Microsoft Cloud	Jamie Rance	Charlotte, NC	April 8, 2015	n/a
SQL Saturday #375 - Silicon Valley 2015	Ike Ellis	Mountain View, CA	March 28, 2015	n/a
Meetup: Istanbul Azure Meetup	Daron Yondem	Istanbul, Turkey	March 7, 2015	n/a
Meetup: Great Lakes Area .Net User Group	Michael Collier	Southfield, MI	February 18, 2015	n/a




EVENT NAME	SPEAKER	LOCATION	DATE	RECORDING
TechX Azure	Magnus Mårtensson	Stockholm, Sweden	January 28-29, 2015	DocumentDB in Azure the new NoSQL option for the Cloud

Videos and Podcasts

SHOW	SPEAKER	DATE	EPISODE
Azure Friday	Kirill Gavrylyuk	October 31, 2016	What's new in Azure DocumentDB?
Channel 9: Microsoft + Open Source	Jose Miguel Parrella	April 14, 2016	From MEAN to DEAN in Azure with Bitnami, VM Scale Sets and DocumentDB
Wired2WinWebinar	Sai Sankar Kunnathukuzhiyil	March 9, 2016	Developing Solutions with Azure DocumentDB
Integration User Group	Han Wong	February 17, 2016	Analyze and visualize non-relational data with DocumentDB + Power BI
The Azure Podcast	Cale Teeter	January 14, 2016	Episode 110: Using DocumentDB & Search
Channel 9: Modern Applications	Tara Shankar Jana	December 13, 2016	Take a modern approach to data in your apps
NinjaTips	Miguel Quintero	December 10, 2015	DocumentDB - Un vistazo general
Integration User Group	Howard Edidin	November 9, 2015	Azure DocumentDB for Healthcare Integration – Part 2
Integration User Group	Howard Edidin	October 5, 2015	Azure DocumentDB for Healthcare Integration
DX Italy - #TechHeroes	Alessandro Melchiori	October 2, 2015	#TechHeroes - DocumentDB
Microsoft Cloud Show - Podcast	Andrew Liu	September 30, 2015	Episode 099 - Azure DocumentDB with Andrew Liu
.NET Rocks! - Podcast	Ryan Crawcour	September 29, 2015	Data on DocumentDB with Ryan CrawCour
Data Exposed	Ryan Crawcour	September 28, 2015	What's New with Azure DocumentDB Since GA
The Azure Podcast	Cale Teeter	September 17, 2015	Episode 94: azpodcast.com re-architecture

SHOW	SPEAKER	DATE	EPISODE
Cloud Cover	Ryan Crawcour	September 4, 2015	Episode 185: DocumentDB Updates with Ryan CrawCour
CodeChat 033	Greg Doerr	July 28, 2015	Greg Doerr on Azure DocumentDB
NoSql Central	King Wilder	May 25, 2015	Golf Tracker - A video overview on how to build a web application on top of AngularJS, WebApi 2, and DocumentDB.
In-Memory Technologies PASS Virtual Chapter	Stephen Baron	May 25, 2015	Hello DocumentDB
Data Exposed	Ryan Crawcour	April 8, 2015	DocumentDB General Availability and What's New!
Data Exposed	Andrew Liu	March 17, 2015	Java SDK for DocumentDB
#DevHangout	Gustavo Alzate Sandoval	March 11, 2015	DocumentDB, la base de datos NoSql de Microsoft Azure
Data Architecture Virtual Chapter PASS	Ike Ellis	February 25, 2015	Introduction to DocumentDB

Online classes



LEARNING PARTNER	DESCRIPTION
	Microsoft Virtual Academy offers you training from the people who help build Azure DocumentDB.
	Pluralsight is a key Microsoft partner offering Azure training. If you are an MSDN subscriber, use your benefits to access Microsoft Azure training.
	OpsGility provides deep technical training on Microsoft Azure. Get instructor-led training on-site or through a remote classroom by their industry-acknowledged trainers.

Discussion

Twitter

Follow us on twitter [@DocumentDB](#) and stay up to date with the latest conversation on the [#DocumentDB](#) hashtag.

Online forums

FORUM PROVIDER	DESCRIPTION
	A language-independent collaboratively edited question and answer site for programmers. Follow our tag: azure-documentdb
	A good place for support and feedback on Microsoft Azure features and services like Web Sites, DocumentDB, etc.

Contact the team



Do you need technical help? Have questions? Wondering whether NoSQL is a good fit for you? You can [schedule a 1:1 chat directly with the DocumentDB engineering team](#). You can also shoot us an [e-mail](#) or tweet us at [@DocumentDB](#).

Open source projects

These projects are actively developed by the Azure DocumentDB team in collaboration with our open source community.

SDKs

PLATFORM	GITHUB	PACKAGE
Node.js	azure-documentdb-node	npm
Java	azure-documentdb-java	Maven
Python	azure-documentdb-python	PyPI




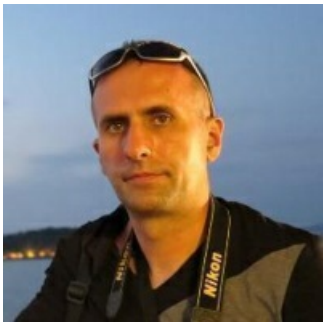
Other projects





NAME	GITHUB	WEBSITE
Documentation	azure-content	Documentation website

NAME	GITHUB	WEBSITE
Hadoop Connector	azure-documentdb-hadoop	Maven
Data migration tool	azure-documentdb-datamigrationtool	Microsoft download center

DocumentDB Wizards

DocumentDB Wizards are community leaders who've demonstrated an exemplary commitment to helping others get the most out of their experience with Azure DocumentDB. They share their exceptional passion, real-world knowledge, and technical expertise with the community and with the DocumentDB team.

WIZARD	PICTURE
Allan Mitchell	
Jen Stirrup	
Lenni Lobel	
Mihail Mateev	

WIZARD	PICTURE
Larry Maccherone	
Howard Edidin	
Santosh Hari	
Matías Quaranta	

Want to become a DocumentDB Wizard? While there is no benchmark for becoming a DocumentDB Wizard, some of the criteria we evaluate include the impact of a nominee's contributions to online forums such as StackOverflow and MSDN; wikis and online content; conferences and user groups; podcasts, Web sites, blogs and social media; and articles and books. You can nominate yourself or someone else by [sending us an email](#).