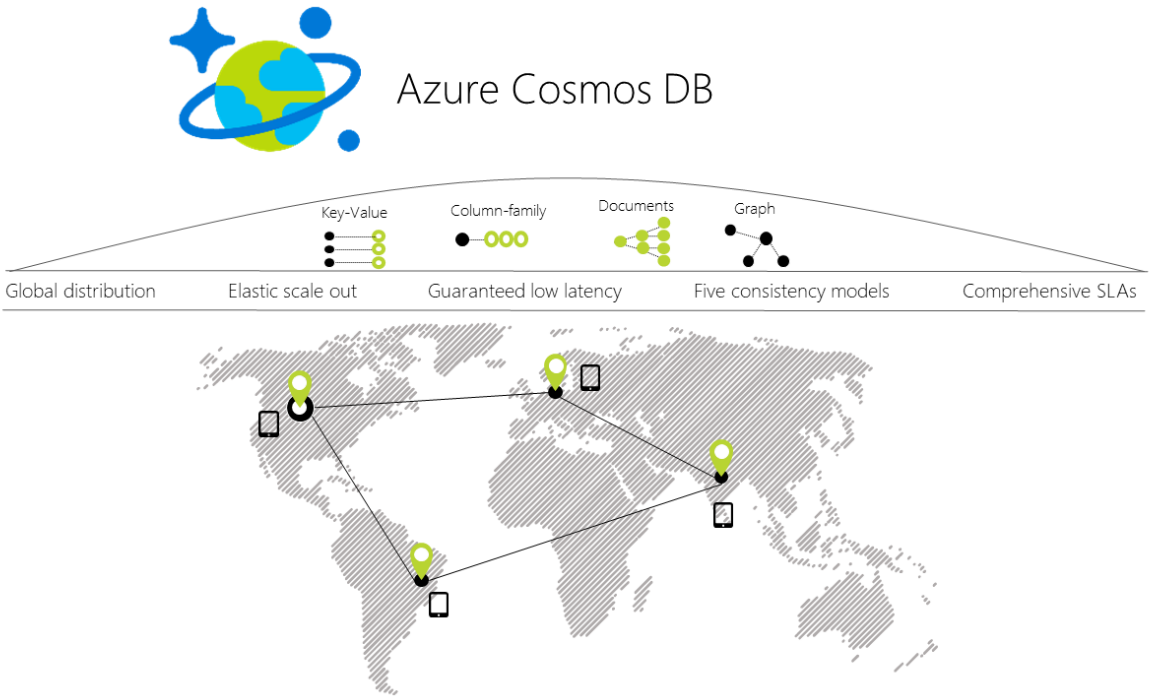
Table of Contents

Welcome to Azure Cosmos DB

6/1/2017 • 9 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed, multi-model database. With the click of a button, Azure Cosmos DB enables you to elastically and independently scale throughput and storage across any number of Azure's geographic regions. It offers throughput, latency, availability, and consistency guarantees with comprehensive service level agreements (SLAs), something no other database service can offer.



Azure Cosmos DB contains a write optimized, resource governed, schema-agnostic database engine that natively supports multiple data models: key-value, documents, graphs, and columnar. It also supports many APIs for accessing data including MongoDB, DocumentDB SQL, Gremlin (preview), and Azure Tables (preview), in an extensible manner.

Azure Cosmos DB started in late 2010 to address developer pain-points that are faced by large scale applications inside Microsoft. Since building globally distributed applications is not a problem unique to just to Microsoft, we made the service available externally to all Azure Developers in the form of Azure DocumentDB. Azure Cosmos DB is the next big leap in the evolution of DocumentDB and we are now making it available for you to use. As a part of this release of Azure Cosmos DB, DocumentDB customers (with their data) are automatically Azure Cosmos DB customers. The transition is seamless and they now have access to a broader range of new capabilities offered by Azure Cosmos DB.

Capability comparison

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

CAPABILITIES	RELATIONAL DBS	NON-RELATIONAL (NOSQL) DBS	AZURE COSMOS DB
Global distribution	x	x	□ Turnkey, 30+ regions, multi-homing

CAPABILITIES	RELATIONAL DBS	NON-RELATIONAL (NOSQL) DBS	AZURE COSMOS DB
Horizontal scale	x	□	□ Independently scale storage and throughput
Latency guarantees	x	□	□ <10 ms for reads, <15 ms for writes at p99
High availability	x	□	□ Always on, PACELC tradeoffs, automatic & manual failover
Data model + API	Relational + SQL	Multi-model + OSS API	Multi-model + SQL + OSS API (more coming soon)
SLAs	□	x	□ Comprehensive SLAs for latency, throughput, consistency, availability

Key capabilities

As a globally distributed database service, Azure Cosmos DB provides the following capabilities to help you build scalable, globally distributed, highly responsive applications:

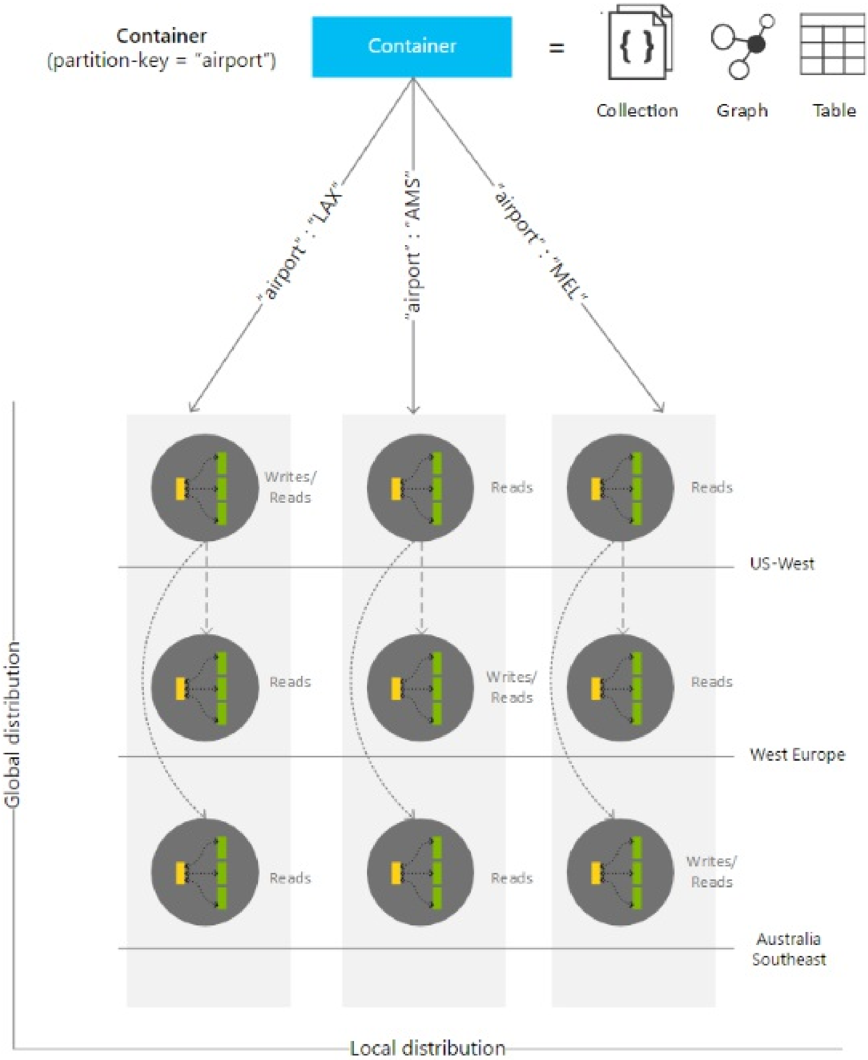
- **Turnkey global distribution**
  - Your application is instantly available to your users, everywhere. Now your data can be too.
  - Don't worry about hardware, adding nodes, VMs or cores. Just point and click, and your data is there.
- **Multiple data models and popular APIs for accessing and querying data**
  - Support for multiple data models including key-value, document, graph, and columnar.
  - Extensible APIs for Node.js, Java, .NET, .NET Core, Python, and MongoDB.
  - SQL and Gremlin for queries.
- **Elastically scale throughput and storage on demand, worldwide**
  - Easily scale throughput at second and minute granularities, and change it anytime you want.
  - Scale storage transparently and automatically to cover your size requirements now and forever.
- **Build highly responsive and mission-critical applications**
  - Get access to your data with single digit millisecond latencies at the 99th percentile, anywhere in the world.
- **Ensure "always on" availability**
  - 99.99% availability within a single region.
  - Deploy to any number of Azure regions for higher availability.
  - Simulate a failure of one or more regions with zero-data loss guarantees.
- **Write globally distributed applications, the right way**
  - Five consistency models models offer strong SQL-like consistency to NoSQL-like eventual consistency, and every thing in between.
- **Money back guarantees**
  - Your data gets there fast, or your money back.
  - Service level agreements for availability, latency, throughput, and consistency.

- **No database schema/index management**
  - Stop worrying about keeping your database schema and indexes in-sync with your application's schema. We're schema-free.
- **Low cost of ownership**
  - Five to ten times more cost effective than a non-managed solution.
  - Three times cheaper than DynamoDB.

Global distribution

Azure Cosmos DB containers are distributed along two dimensions:

1. Within a given region, all resources are horizontally partitioned using resource partitions (local distribution).
2. Each resource partition is also replicated across geographical regions (global distribution).



When your storage and throughput needs to be scaled, Cosmos DB transparently performs partition management operations across all the regions. Independent of the scale, distribution, or failures, Cosmos DB continues to provide a single system image of the globally distributed resources.

Global distribution of resources in Cosmos DB is turn-key. At any time with a few button clicks (or programmatically with a single API call), you can associate any number of geographical regions with your database account.

Regardless of the amount of data or the number of regions, Cosmos DB guarantees each newly associated region to start processing client requests under an hour at the 99th percentile. This is done by parallelizing the seeding and copying data from all the source resource partitions to the newly associated region. Customers can also remove an existing region or take a region that was previously associated with their database account offline.

Multi-model, multi-API support

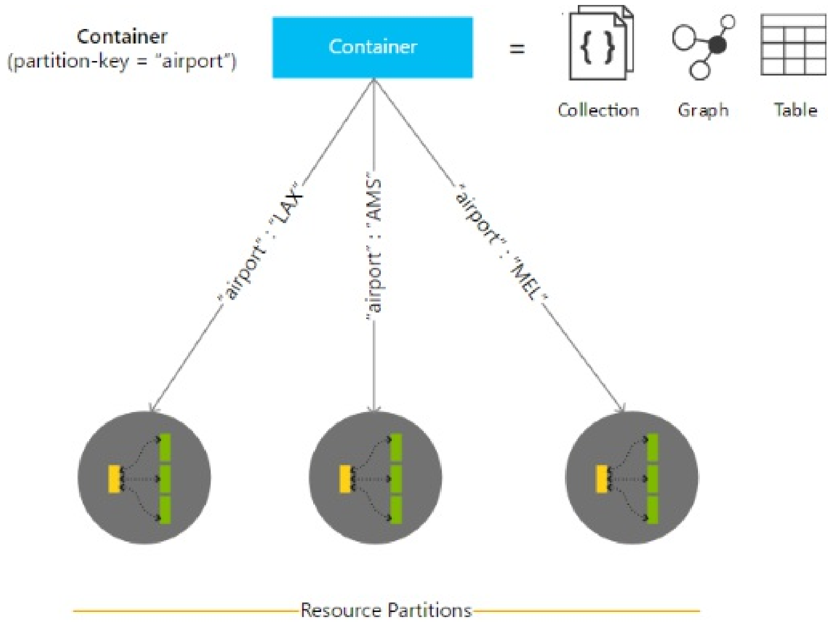
Azure Cosmos DB natively supports multiple data models including documents, key-value, graph, and column-family. The core content-model of Cosmos DB's database engine is based on atom-record-sequence (ARS). Atoms consist of a small set of primitive types like string, bool, and number. Records are structs composed of these types. Sequences are arrays consisting of atoms, records, or sequences.

The database engine can efficiently translate and project different data models onto the ARS-based data model. The core data model of Cosmos DB is natively accessible from dynamically typed programming languages and can be exposed as-is as JSON.

The service also supports popular database APIs for data access and querying. Cosmos DB's database engine currently supports DocumentDB SQL, MongoDB, Azure Tables (preview), and Gremlin (preview). You can continue to build applications using popular OSS APIs and get all the benefits of a battle-tested and fully managed, globally distributed database service.

Horizontal scaling of storage and throughput

All the data within a Cosmos DB container (for example, a document collection, table, or graph) is horizontally partitioned and transparently managed by resource partitions. A resource partition is a consistent and highly available container of data partitioned by a customer specified partition-key. It provides a single system image for a set of resources it manages and is a fundamental unit of scalability and distribution. Cosmos DB is designed to let you elastically scale throughput based on the application traffic patterns across different geographical regions to support fluctuating workloads varying both by geography and time. The service manages the partitions transparently without compromising the availability, consistency, latency, or throughput of a Cosmos DB container.



You can elastically scale throughput of an Azure Cosmos DB container by programmatically provisioning throughput using request units per second (RU/s). Internally, the service transparently manages resource partitions to deliver the throughput on a given container. Cosmos DB ensures that the throughput is available for use across all the regions associated with the container. The new throughput is effective within five seconds of the change in the configured throughput value.

You can provision throughput on a Cosmos DB container at both, per-second and at per-minute (RU/m) granularities. The provisioned throughput at per-minute granularity is used to manage unexpected spikes in the workload occurring at a per-second granularity.

Low latency guarantees at the 99th percentile

As part of its SLAs, Cosmos DB guarantees end-to-end low latency at the 99th percentile to its customers. For a typical 1-KB item, Cosmos DB guarantees end-to-end latency of reads under 10 ms and indexed writes under 15 ms at the 99th percentile, within the same Azure region. The median latencies are significantly lower (under 5 ms). With an upper bound of request processing on every database transaction, Cosmos DB allows clients to clearly distinguish between transactions with high latency vs. a database being unavailable.

Transparent multi-homing and 99.99% high availability

You can dynamically associate "priorities" to the regions associated with your Azure Cosmos DB database account. Priorities are used to direct the requests to specific regions in the event of regional failures. In an unlikely event of a regional disaster, Cosmos DB automatically failovers in the order of priority.

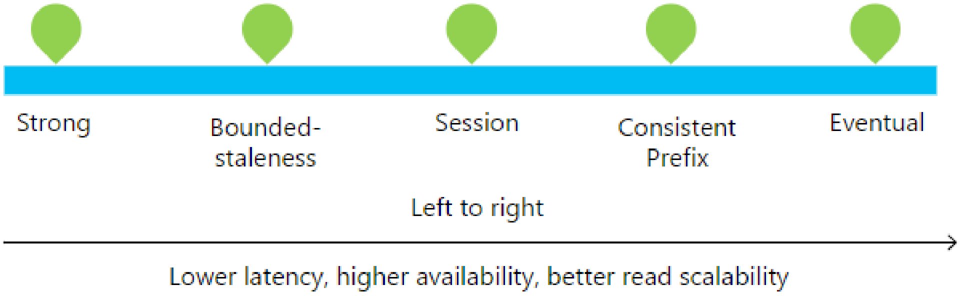
To test the end-to-end availability of the application, you can manually trigger failover (rate limited to two operations within an hour). Cosmos DB guarantees zero data loss during manual regional failovers. In case a regional disaster occurs, Cosmos DB guarantees an upper-bound on data loss during the system-initiated automatic failover. You do not have to redeploy your application after a regional failover, and availability SLAs are maintained by Azure Cosmos DB.

For this scenario, Cosmos DB allows you to interact with resources using either logical (region-agnostic) or physical (region-specific) endpoints. The former ensures that the application can transparently be multi-homed in case of failover. The latter provides fine-grained control to the application to redirect reads and writes to specific regions. Cosmos DB guarantees 99.99% availability SLA for every database account. The availability guarantees are agnostic of the scale (provisioned throughput and storage), number of regions, or geographical distance between regions associated with a given database.

Multiple, well-defined consistency models

Commercial distributed databases fall into two categories: databases that do not offer well-defined, provable consistency choices at all, and databases which offer two extreme programmability choices (strong vs. eventual consistency). The former burdens application developers with minutia of their replication protocols and expects them to make difficult tradeoffs between consistency, availability, latency, and throughput. The latter puts a pressure to choose one of the two extremes. Despite the abundance of research and proposals for more than 50 consistency models, the distributed database community has not been able to commercialize consistency levels beyond strong and eventual consistency.

Cosmos DB allows you to choose between five well-defined consistency models along the consistency spectrum – strong, bounded staleness, session, consistent prefix, and eventual.



The following table illustrates the specific guarantees each consistency level provides.

Consistency Levels and guarantees

CONSISTENCY LEVEL	GUARANTEES
Strong	Linearizability
Bounded Staleness	Consistent Prefix. Reads lag behind writes by k prefixes or t interval
Session	Consistent Prefix. Monotonic reads, monotonic writes, read-your-writes, write-follows-reads
Consistent Prefix	Updates returned are some prefix of all the updates, with no gaps
Eventual	Out of order reads

You can configure the default consistency level on your Cosmos DB account (and later override the consistency on a specific read request). Internally, the default consistency level applies to data within the partition sets which may be span regions.

Guaranteed service level agreements

Cosmos DB is the first managed database service to offer 99.99% SLA guarantees for availability, throughput, low latency, and consistency.

- Availability: 99.99% uptime availability SLA for each of the data and control plane operations.
- Throughput: 99.99% of requests complete successfully
- Latency: 99.99% of <10 ms latencies at the 99th percentile
- Consistency: 100% of read requests will meet the consistency guarantee for the consistency level requested by you.

Schema-free

Both relational and NoSQL databases force you to deal with schema & index management, versioning and migration – all of this is extremely challenging in a globally distributed setup. But don’t worry -- Cosmos DB makes this problem go away! With Cosmos DB, you do not have to manage schemas and indexes, deal with schema versioning or worry about application downtime while migrating schemas. Cosmos DB’s database engine is fully schema-agnostic – it automatically indexes all the data it ingests without requiring any schema or indexes and serves blazing fast queries.

Low cost of ownership

When all total cost of ownership (TCO) considerations taken into account, managed cloud services like Azure Cosmos DB can be five to ten times more cost effective than their OSS counter-parts running on-premises or virtual machines. And Azure Cosmos DB is up to two to three times cheaper than DynamoDB for high volume workloads. Learn more in the TCO whitepaper.

Next steps

Get started with Azure Cosmos DB with one of our quickstarts:

- Get started with Azure Cosmos DB's DocumentDB API
- Get started with Azure Cosmos DB's MongoDB API
- Get started with Azure Cosmos DB's Graph API

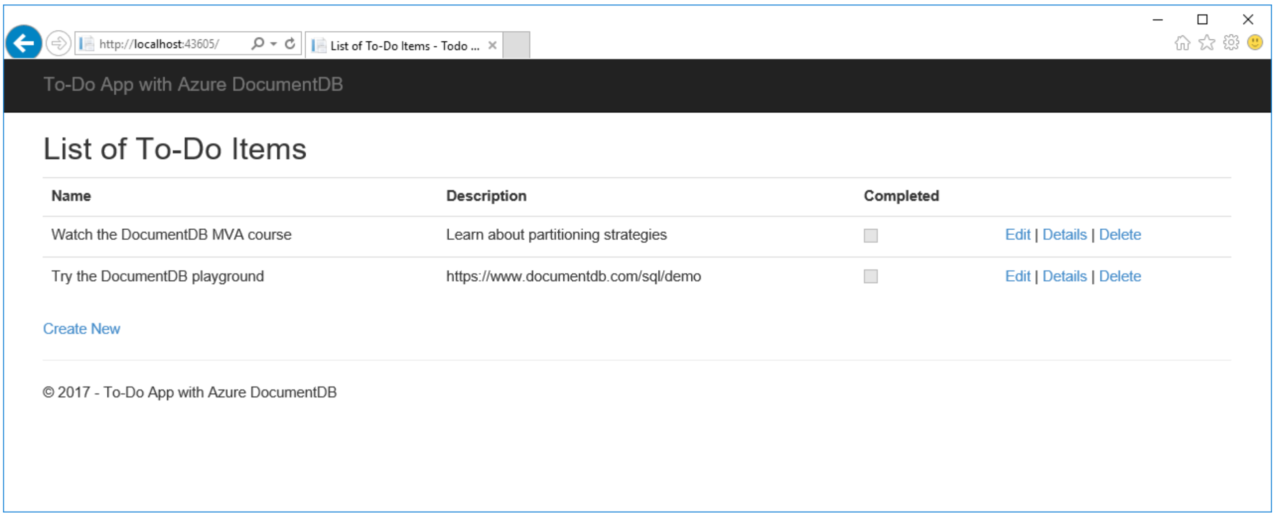
- Get started with Azure Cosmos DB's Table API

Azure Cosmos DB: Build a DocumentDB API web app with .NET and the Azure portal

6/1/2017 • 7 min to read • Edit Online

Azure Cosmos DB is Microsoft’s globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quick start demonstrates how to create an Azure Cosmos DB account, document database, and collection using the Azure portal. You'll then build and deploy a todo list web app built on the DocumentDB .NET API, as shown in the following screenshot.


http://localhost:43605/
List of To-Do Items - Todo ...
To-Do App with Azure DocumentDB

List of To-Do Items

Name	Description	Completed	
Watch the DocumentDB MVA course	Learn about partitioning strategies	□	Edit | Details | Delete
Try the DocumentDB playground	https://www.documentdb.com/sql/demo	□	Edit | Details | Delete

Create New

© 2017 - To-Do App with Azure DocumentDB

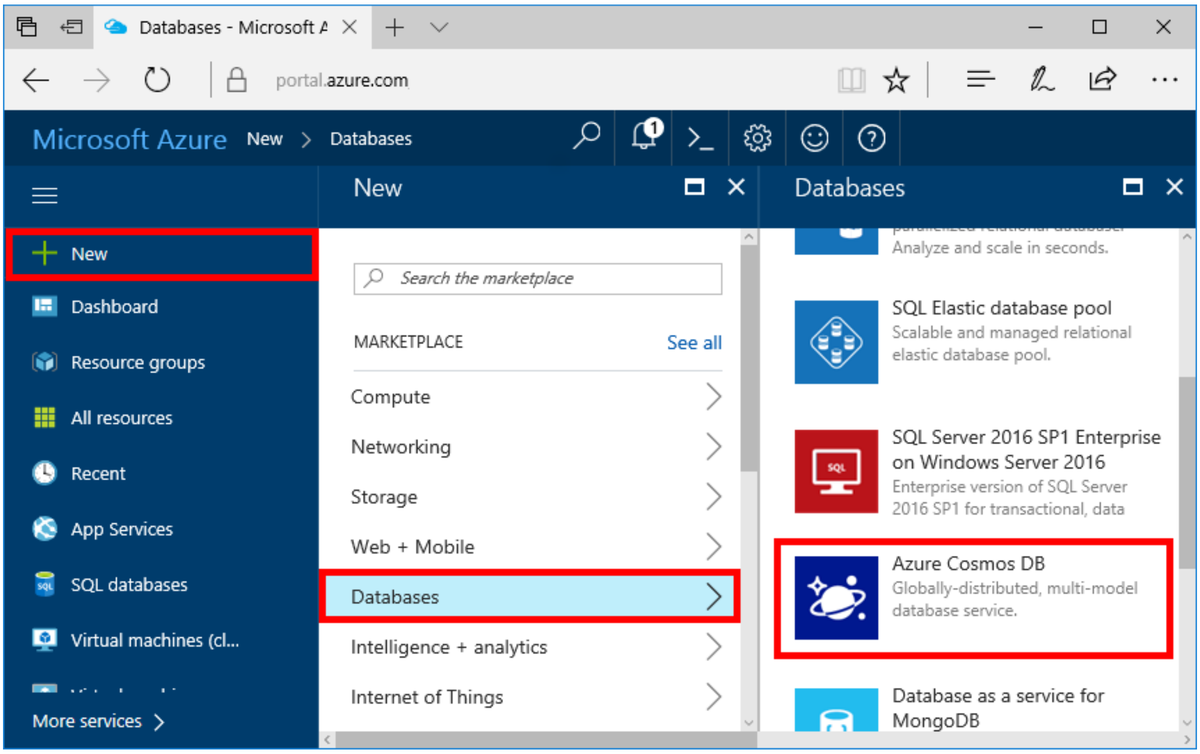
Prerequisites

If you don’t already have Visual Studio 2017 installed, you can download and use the **free** Visual Studio 2017 Community Edition. Make sure that you enable **Azure development** during the Visual Studio setup.

If you don't have an Azure subscription, create a free account before you begin.

Create a database account

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

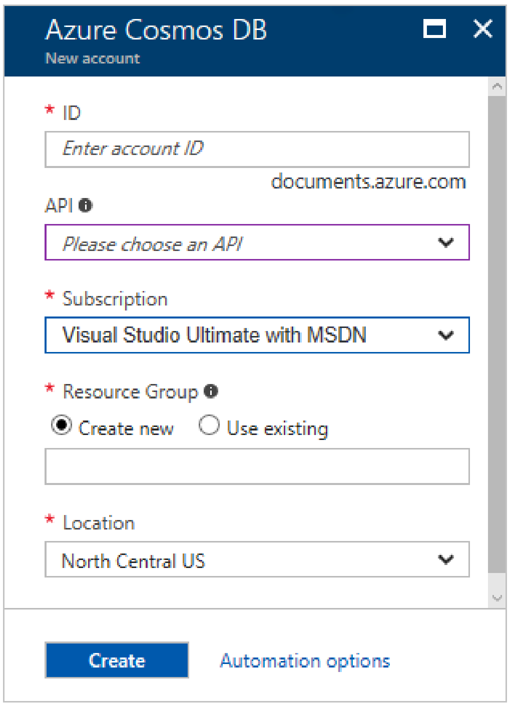


3. On the **New account** blade, specify the configuration that you want for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article we program against the DocumentDB API, so choose **SQL (DocumentDB)** as you fill out the form. But if you have graph data for a social media app, or key/value (table) data, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

   Complete the fields on the **New account** blade, using the information in the following screenshot as a guide. When you set up your account, choose unique values that do not match those in the screenshot.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

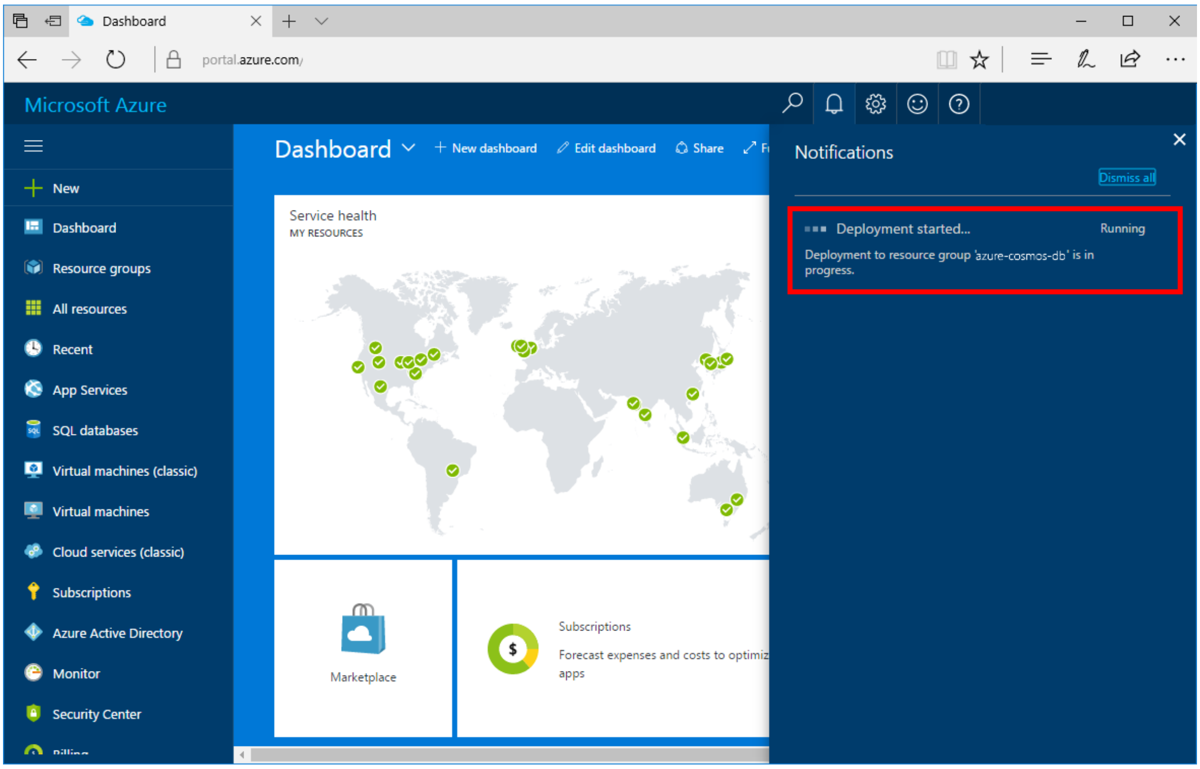
* Subscription
Visual Studio Ultimate with MSDN

* Resource Group
● Create new ○ Use existing

* Location
North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that identifies your Azure Cosmos DB account. The string *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 through 50 characters.
API	SQL (DocumentDB)	We program against the DocumentDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for your Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource-group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location that's closest to your users to give them the fastest access to the data.

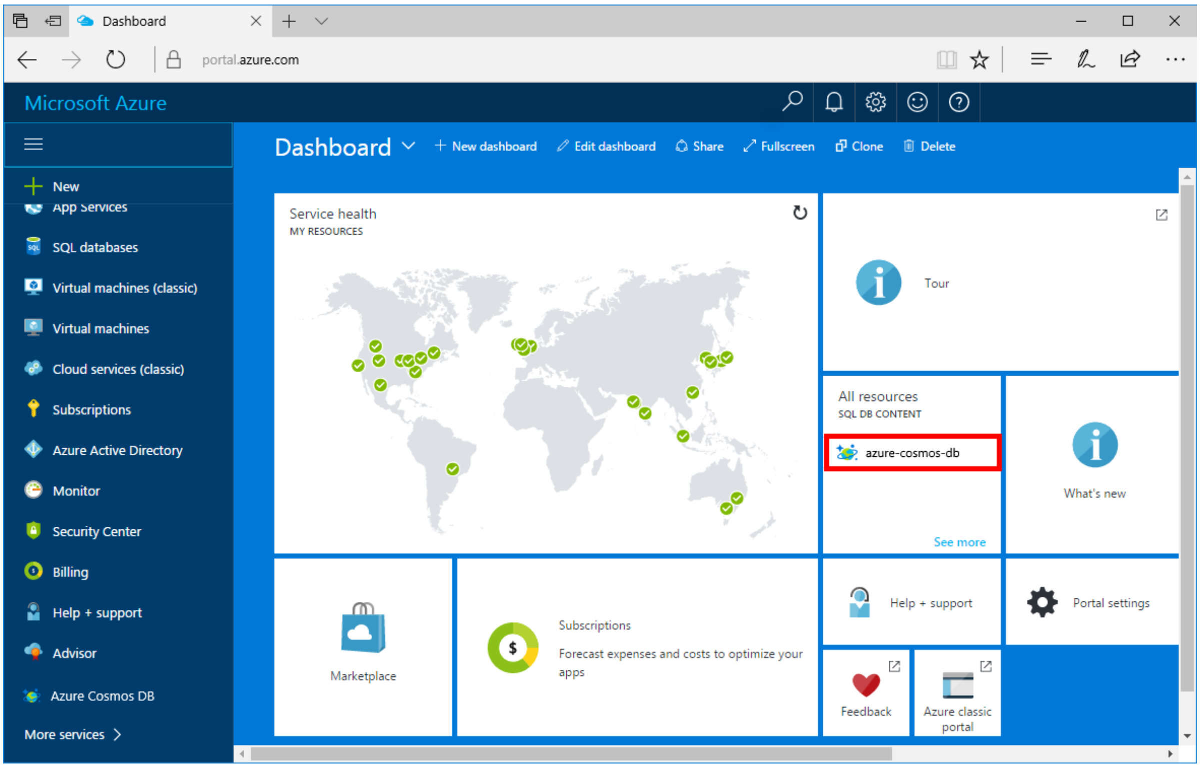
4. Click **Create** to create the account.
5. On the top toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

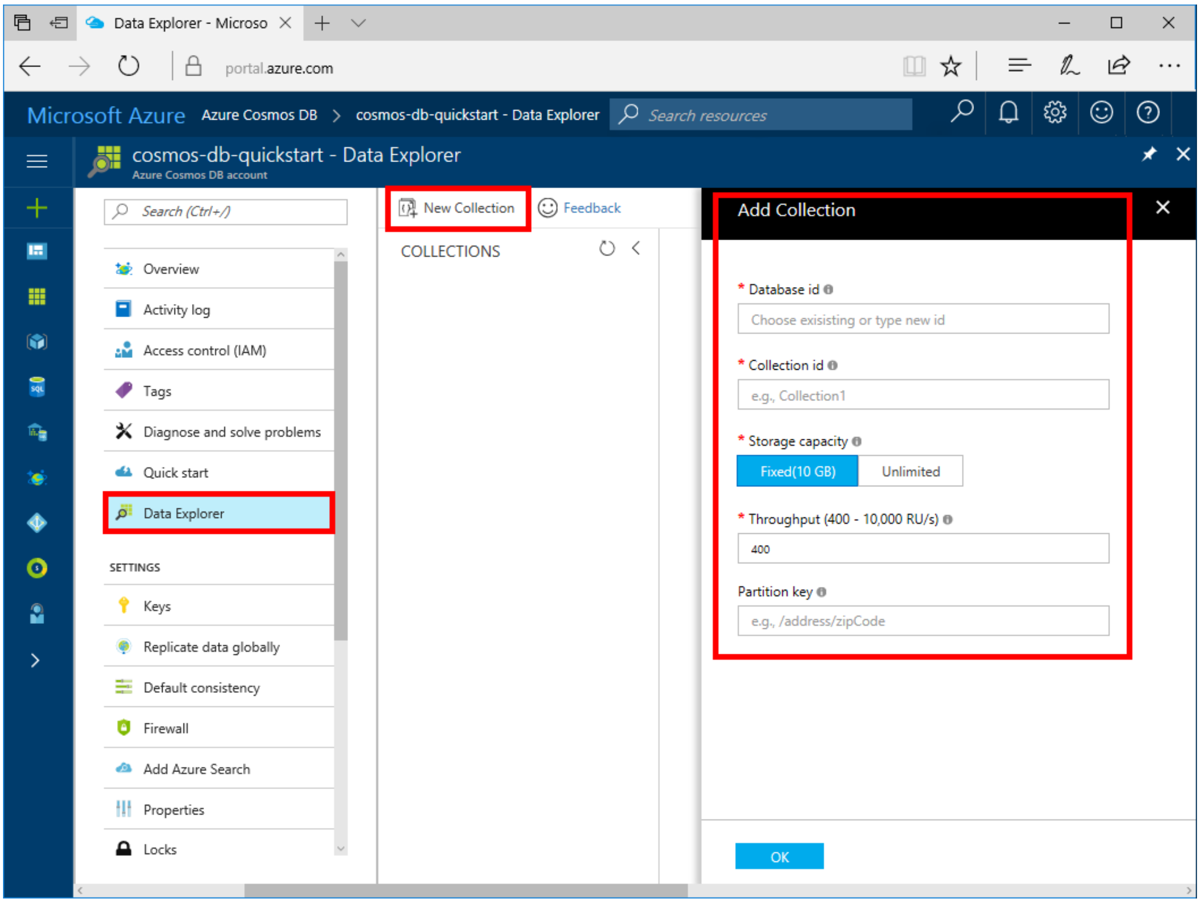
6. When the deployment is complete, open the new account from the **All Resources** tile.



Add a collection

You can now use Data Explorer to create a collection and add data to your database.

1. In the Azure portal, in the left pane, click **Data Explorer**.
2. On the **Data Explorer** blade, click **New Collection**, and then provide the following information:



Add Collection

* Database id

Choose exisisting or type new id

* Collection id

e.g., Collection1

* Storage capacity

Fixed(10 GB) | Unlimited

* Throughput (400 - 10,000 RU/s)

400

Partition key

e.g., /address/zipCode

OK

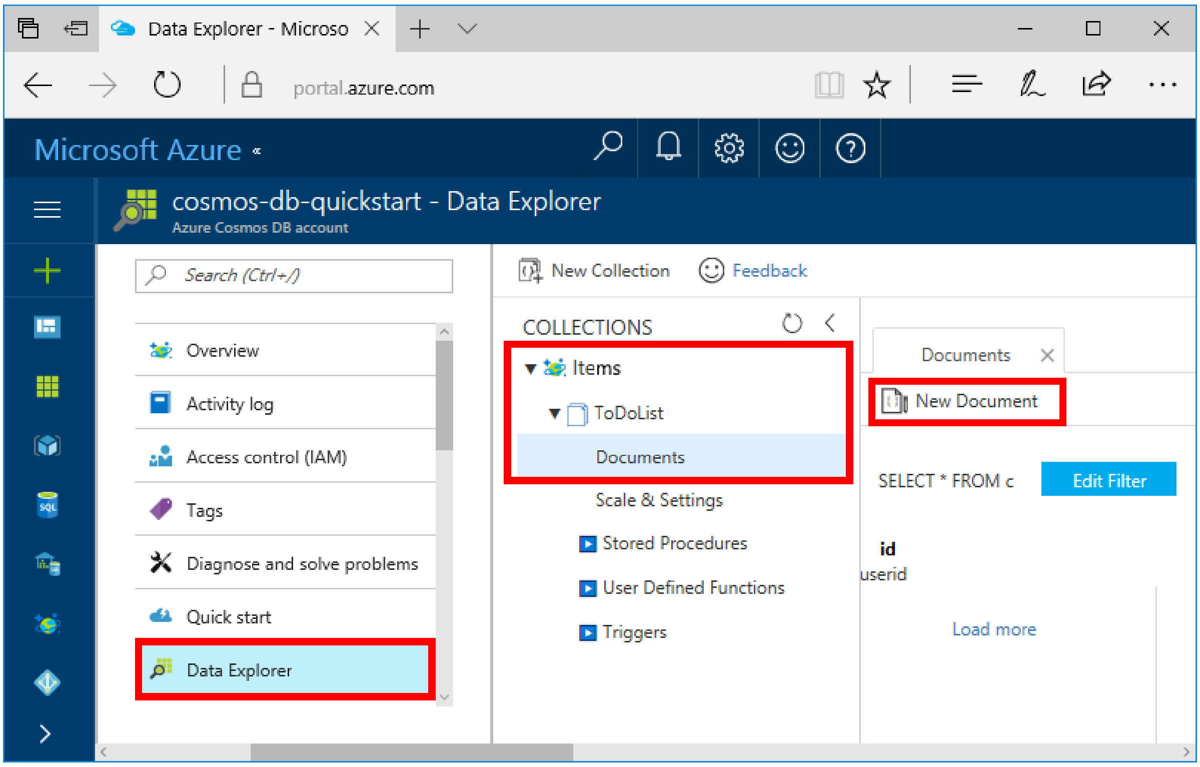
SETTING	SUGGESTED VALUE	DESCRIPTION
Database id	Items	The ID for your new database. Database names must contain from 1 through 255 characters, and they cannot contain /, \, #, ?, or a trailing space.
Collection id	ToDoList	The ID for your new collection. Collection names have the same character requirements as database IDs.
Storage capacity	Fixed (10 GB)	Use the default value. This is the storage capacity of the database.
Throughput	400 RU	Use the default value. If you want to reduce latency, you can scale up the throughput later.
Partition key	/userid	A partition key that distributes data evenly to each partition. Selecting the correct partition key is important in creating a performant collection. To learn more, see Designing for partitioning.

3. After you've completed the form, click **OK**.

Add sample data

You can now add data to your new collection using Data Explorer.

1. In Data Explorer, the new database appears in the Collections pane. Expand the **Items** database, expand the **ToDoList** collection, click **Documents**, and then click **New Documents**.



2. Now add a few documents to the collection with the following structure, where you insert unique values for id in each document and change the other properties as you see fit. Your new documents can have any structure you want as Azure Cosmos DB doesn't impose any schema on your data.

```
{
  "id": "1",
  "category": "personal",
  "name": "groceries",
  "description": "Pick up apples and strawberries."
}
```

   You can now use queries in Data Explorer to retrieve your data. By default, Data Explorer uses `SELECT * FROM c` to retrieve all documents in the collection, but you can change that to `SELECT * FROM c ORDER BY c.name ASC`, to return all the documents in alphabetic ascending order of the name property.

   You can also use Data Explorer to create stored procedures, UDFs, and triggers to perform server-side business logic as well as scale throughput. Data Explorer exposes all of the built-in programmatic data access available in the APIs, but provides easy access to your data in the Azure portal.

Clone the sample application

Now let's clone a DocumentDB API app from github, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a git terminal window, such as git bash, and `CD` to a working directory.

2. Run the following command to clone the sample repository.

```
git clone https://github.com/Azure-Samples/documentdb-dotnet-todo-app.git
```

3. Then open the solution file in Visual Studio.

Review the code

Let's make a quick review of what's happening in the app. Open the DocumentDBRepository.cs file and you'll find that these lines of code create the Azure Cosmos DB resources.

- The DocumentClient is initialized.

```
client = new DocumentClient(new Uri(ConfigurationManager.AppSettings["endpoint"]),
ConfigurationManager.AppSettings["authKey"]);`
```

- A new database is created.

```
await client.CreateDatabaseAsync(new Database { Id = DatabaseId });
```

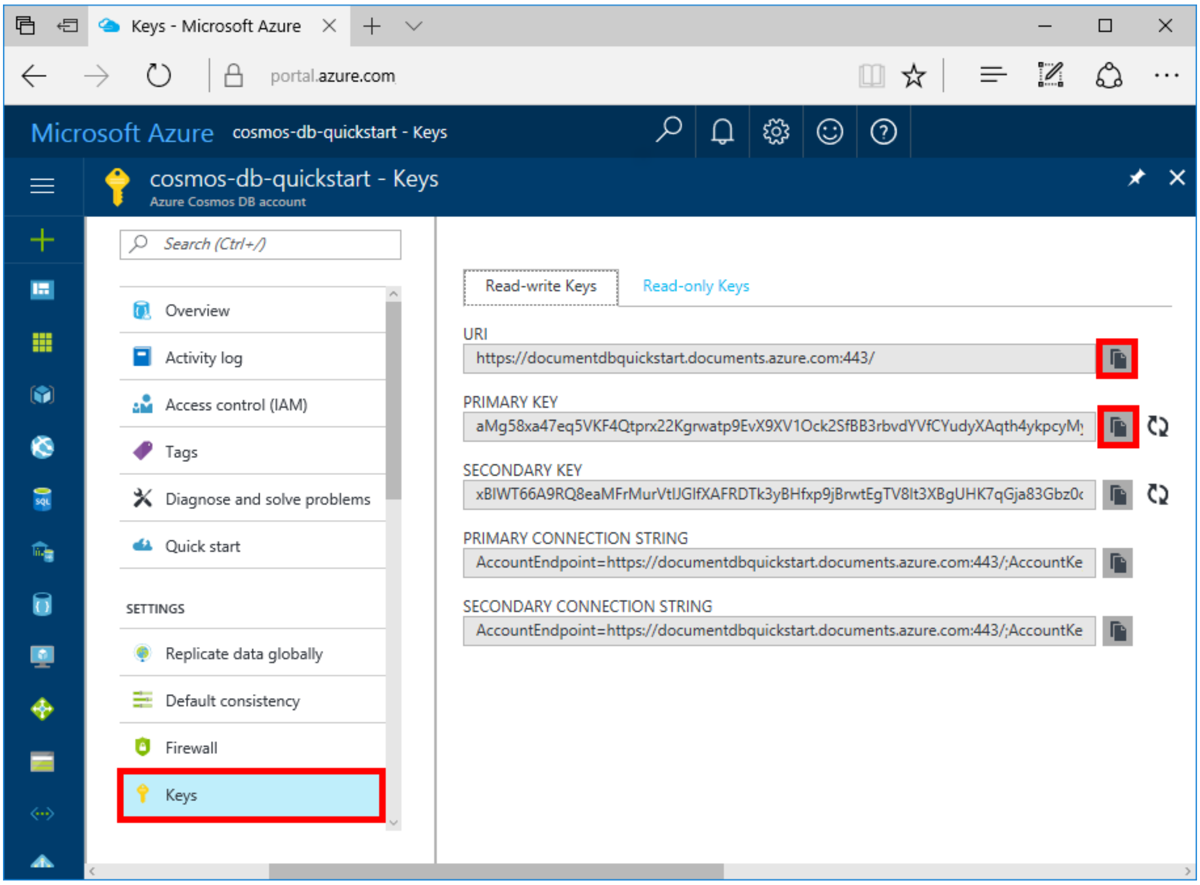
- A new collection is created.

```
await client.CreateDocumentCollectionAsync(
  UriFactory.CreateDatabaseUri(DatabaseId),
  new DocumentCollection { Id = CollectionId },
  new RequestOptions { OfferThroughput = 1000 });
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. In the Azure portal, in your Azure Cosmos DB account, in the left navigation click **Keys**, and then click **Read-write Keys**. You'll use the copy buttons on the right side of the screen to copy the URI and Primary Key into the web.config file in the next step.



URI
https://documentdbquickstart.documents.azure.com:443/

PRIMARY KEY
aMg58xa47eq5VKF4Qtprx22Kgrwatp9EvX9XV1Ock2SfBB3rbvdYVfCYudyXAqth4ykpcyMy

SECONDARY KEY
xBIWT66A9RQ8eaMFrMurVtIJGIfXAFRDTk3yBHfxp9jBrwtEgTV8It3XBgUHK7qGja83Gbz0c

PRIMARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

SECONDARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

2. In Visual Studio 2017, open the web.config file.

3. Copy your URI value from the portal (using the copy button) and make it the value of the endpoint key in web.config.

```
<add key="endpoint" value="FILLME" />
```

4. Then copy your PRIMARY KEY value from the portal and make it the value of the authKey in web.config. You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

```
<add key="authKey" value="FILLME" />
```

Run the web app

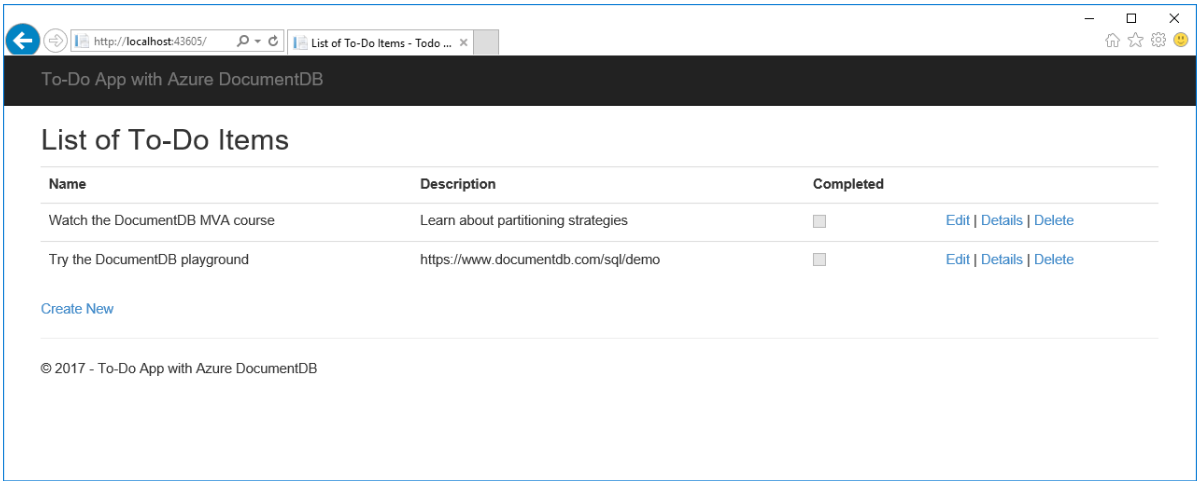
1. In Visual Studio, right-click on the project in **Solution Explorer** and then click **Manage NuGet Packages**.

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

3. From the results, install the **Microsoft.Azure.DocumentDB** library. This installs the Microsoft.Azure.DocumentDB package as well as all dependencies.

4. Click CTRL + F5 to run the application. Your app displays in your browser.

5. Click **Create New** in the browser and create a few new tasks in your to-do app.


http://localhost:43605/
List of To-Do Items - Todo ...
To-Do App with Azure DocumentDB

List of To-Do Items

Name	Description	Completed	
Watch the DocumentDB MVA course	Learn about partitioning strategies	☐	Edit | Details | Delete
Try the DocumentDB playground	https://www.documentdb.com/sql/demo	☐	Edit | Details | Delete

Create New

© 2017 - To-Do App with Azure DocumentDB

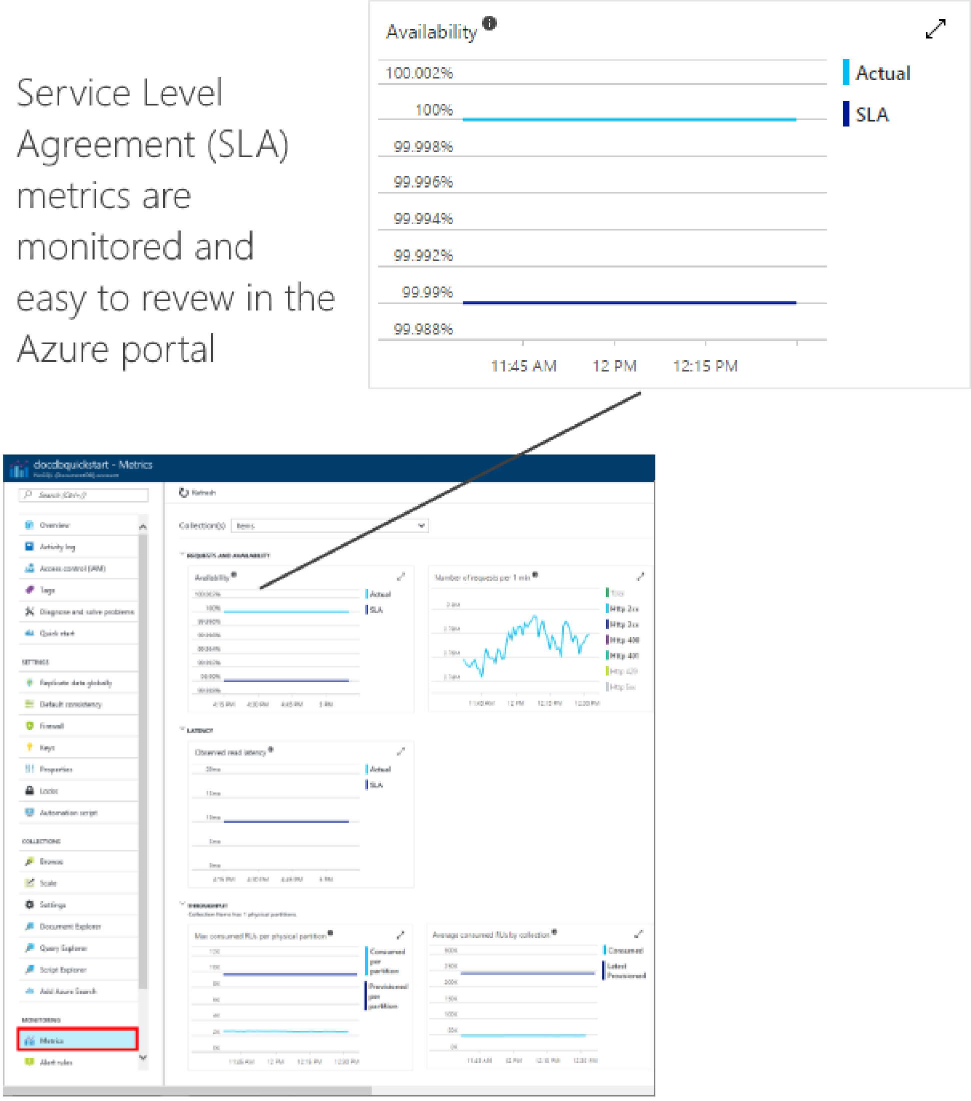
You can now go back to Data Explorer and see query, modify, and work with this new data.

Review SLAs in the Azure portal

Now that your app is up and running, you'll want to ensure business continuity and watch user access to ensure high availability. You can use the Azure portal to review the availability, latency, throughput, and consistency of your collection.

Each graph that's associated with the Azure Cosmos DB Service Level Agreements (SLAs) provides a line that shows the quota required to meet the SLA and your actual usage, giving you a clear view into your database performance. Additional metrics, such as storage usage and number of requests per minute, are also included in the portal.

- In the Azure portal, in the left pane, under **Monitoring**, click **Metrics**.


Service Level Agreement (SLA) metrics are monitored and easy to revew in the Azure portal

Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart in the Azure portal with the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a collection using the Data Explorer, and run a web app. You can now import additional data to your Cosmos DB account.

Import data into Azure Cosmos DB

Azure Cosmos DB: Build a DocumentDB API app with Java and the Azure portal

6/1/2017 • 6 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quick start demonstrates how to create an Azure Cosmos DB account, document database, and collection using the Azure portal. You'll then build and run a console app built on the DocumentDB Java API.

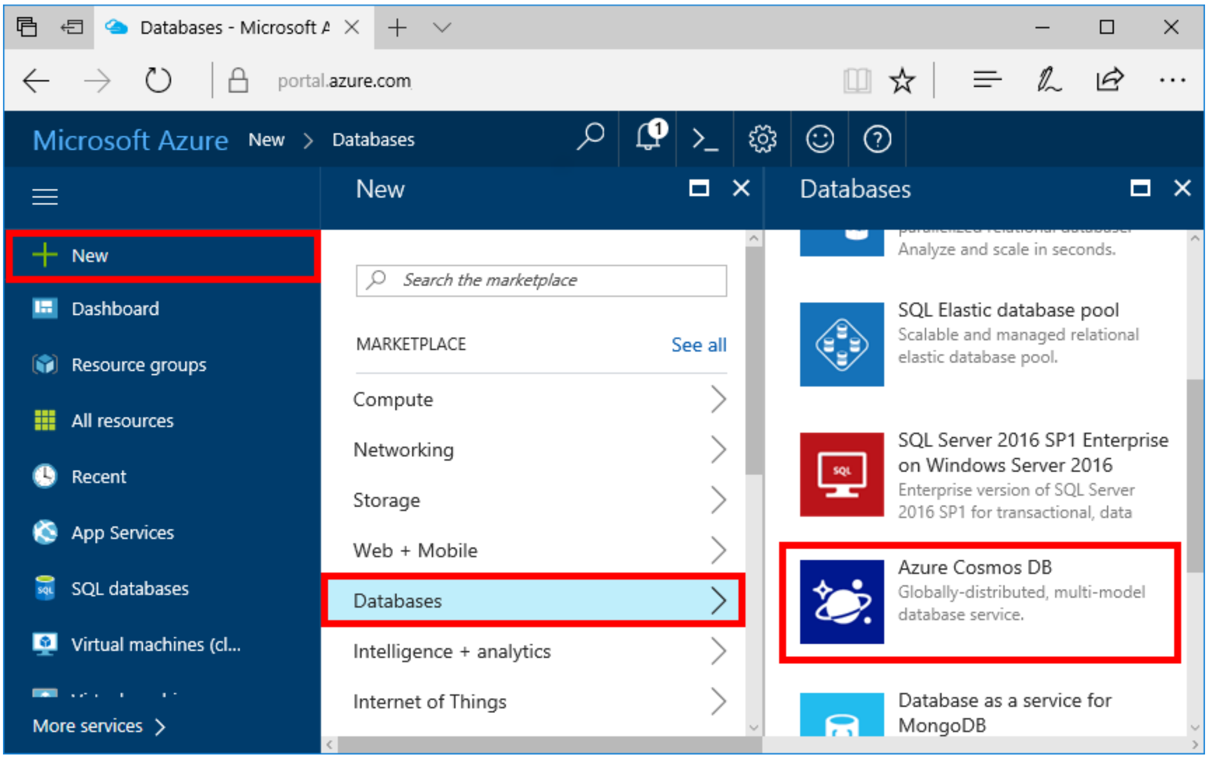
Prerequisites

- Before you can run this sample, you must have the following prerequisites:
  - JDK 1.7+ (Run `apt-get install default-jdk` if you don't have JDK)
  - Maven (Run `apt-get install maven` if you don't have Maven)

If you don't have an Azure subscription, create a free account before you begin.

Create a database account

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.



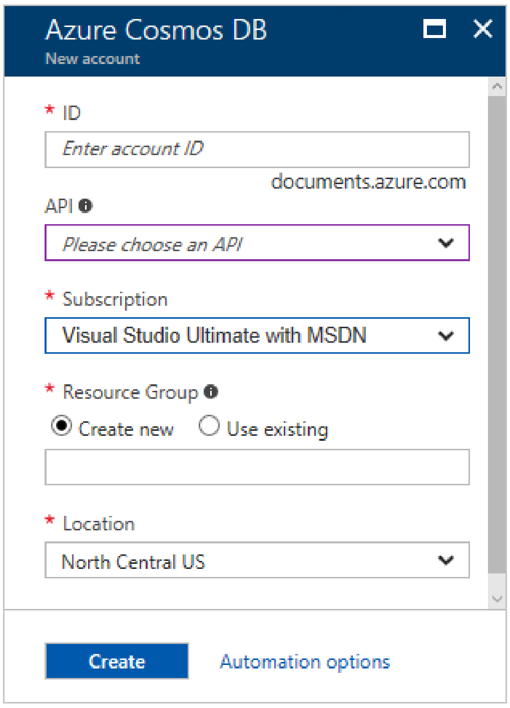
3. On the **New account** blade, specify the configuration that you want for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article we program against the DocumentDB API, so choose **SQL (DocumentDB)** as you

fill out the form. But if you have graph data for a social media app, or key/value (table) data, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

Complete the fields on the **New account** blade, using the information in the following screenshot as a guide. When you set up your account, choose unique values that do not match those in the screenshot.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

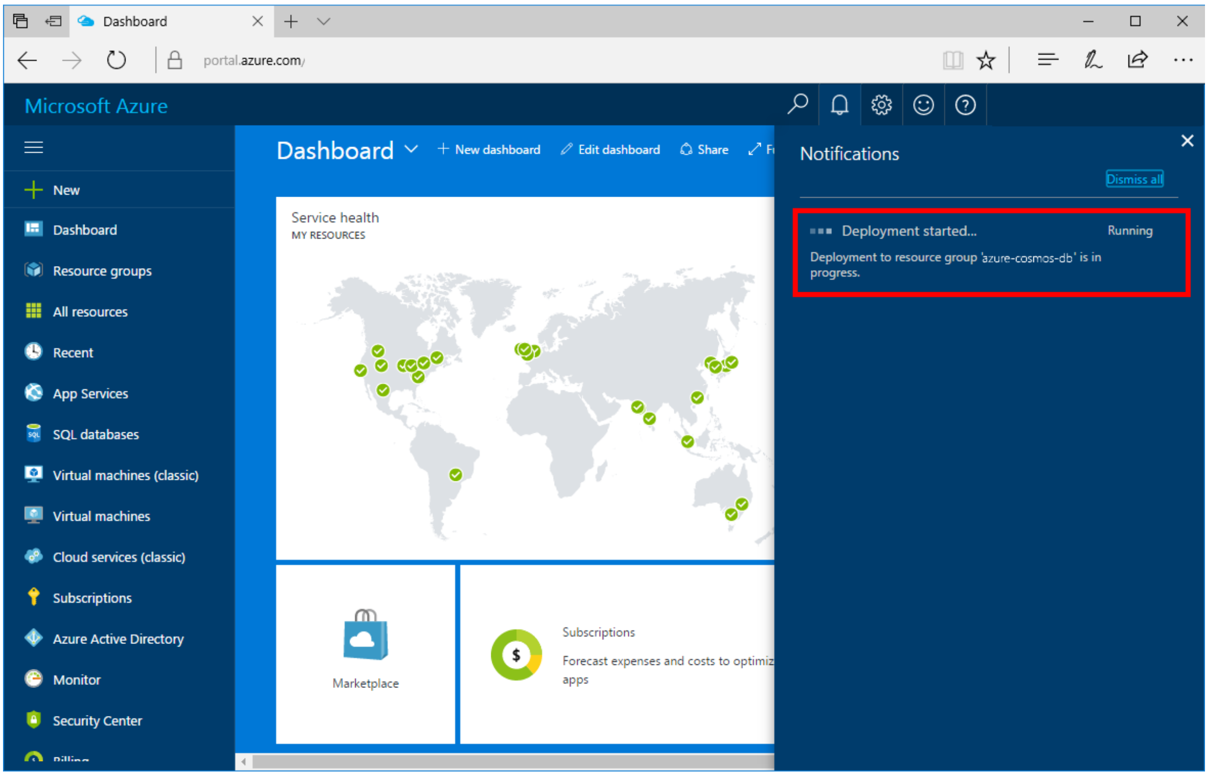
* Subscription
Visual Studio Ultimate with MSDN

* Resource Group
● Create new ○ Use existing

* Location
North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that identifies your Azure Cosmos DB account. The string *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 through 50 characters.
API	SQL (DocumentDB)	We program against the DocumentDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for your Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource-group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location that's closest to your users to give them the fastest access to the data.

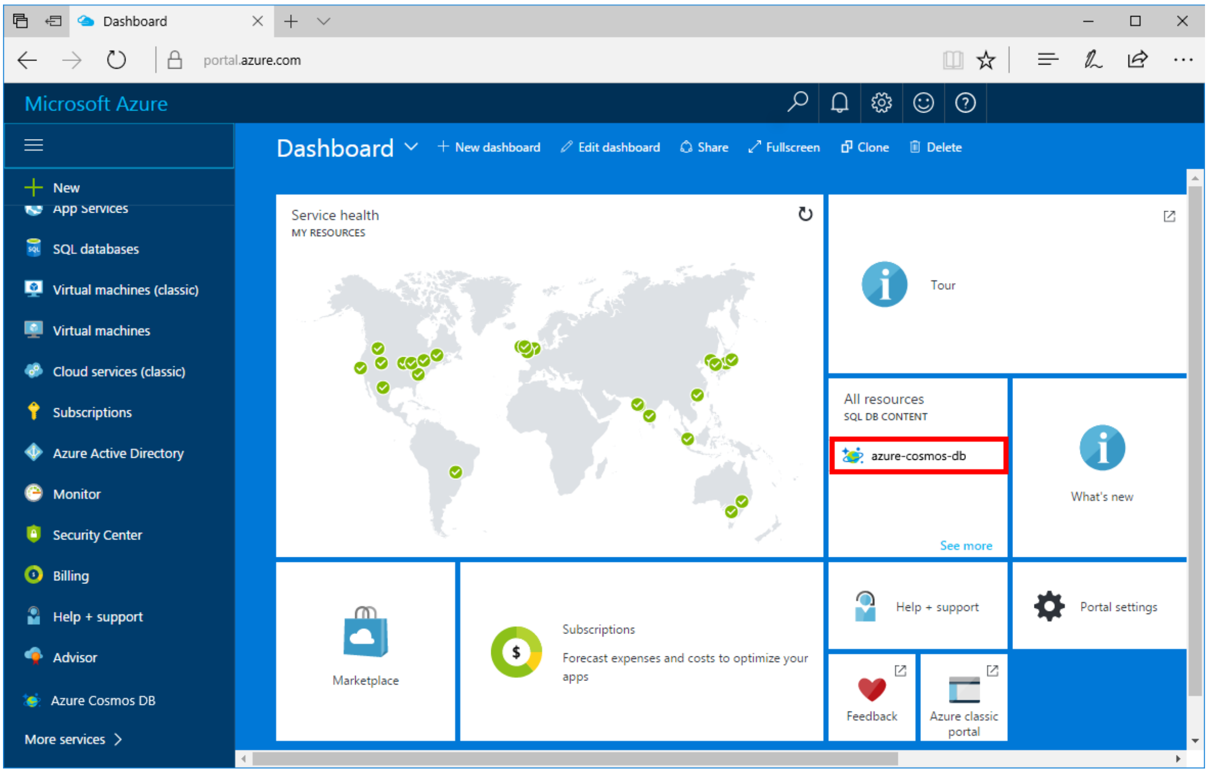
4. Click **Create** to create the account.
5. On the top toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the **All Resources** tile.

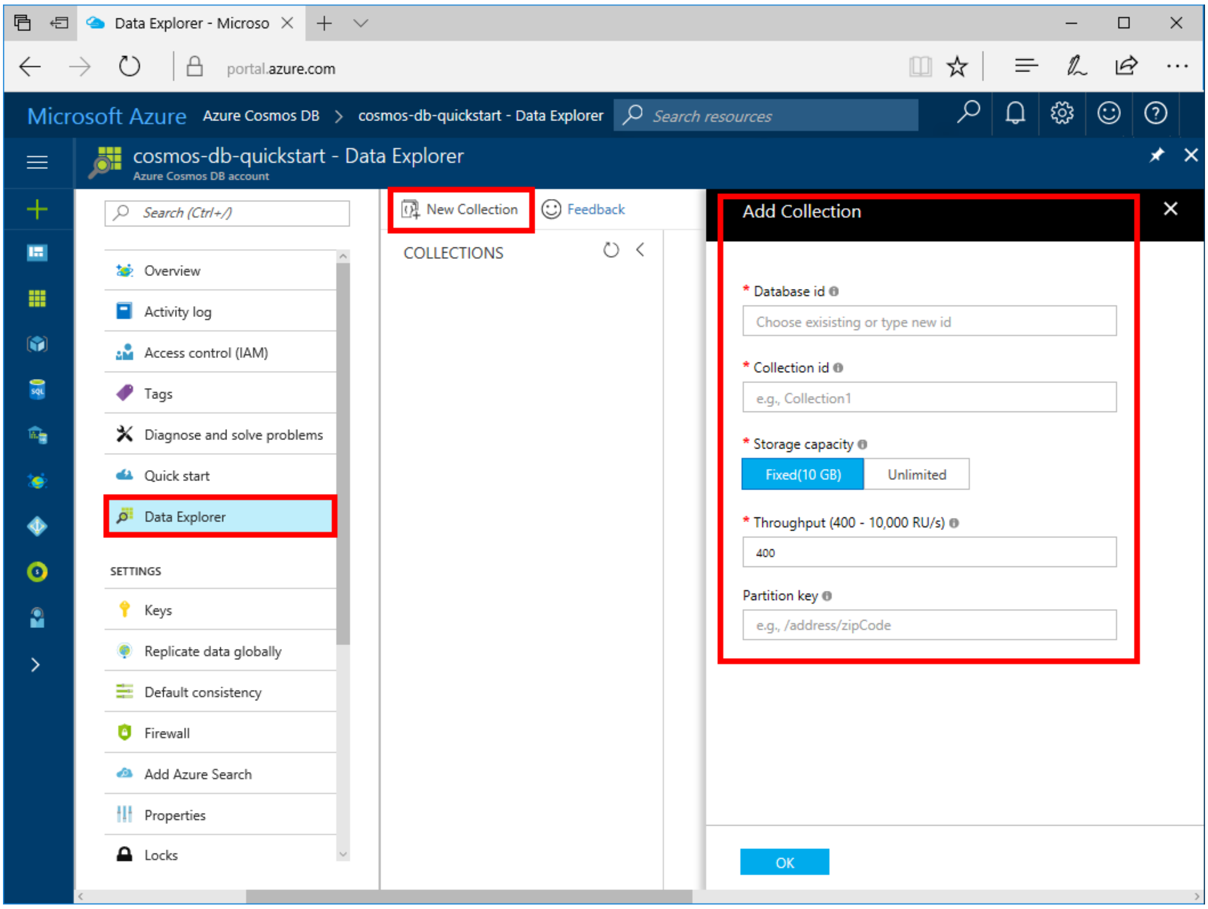


Add a collection

You can now use Data Explorer to create a collection and add data to your database.

1. In the Azure portal, in the left pane, click **Data Explorer**.

2. On the **Data Explorer** blade, click **New Collection**, and then provide the following information:



Add Collection

* Database id
Choose exisisting or type new id

* Collection id
e.g., Collection1

* Storage capacity
● Fixed(10 GB) ○ Unlimited

* Throughput (400 - 10,000 RU/s)
400

Partition key
e.g., /address/zipCode

SETTING	SUGGESTED VALUE	DESCRIPTION
Database id	Items	The ID for your new database. Database names must contain from 1 through 255 characters, and they cannot contain /, \, #, ?, or a trailing space.
Collection id	ToDoList	The ID for your new collection. Collection names have the same character requirements as database IDs.
Storage capacity	Fixed (10 GB)	Use the default value. This is the storage capacity of the database.
Throughput	400 RU	Use the default value. If you want to reduce latency, you can scale up the throughput later.
Partition key	/userid	A partition key that distributes data evenly to each partition. Selecting the correct partition key is important in creating a performant collection. To learn more, see Designing for partitioning.

3. After you've completed the form, click **OK**.

Clone the sample application

Now let's clone a DocumentDB API app from github, set the connection string, and run it. You see how easy it is to

work with data programmatically.

1. Open a git terminal window, such as git bash, and `CD` to a working directory.

2. Run the following command to clone the sample repository.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-java-getting-started.git
```

Review the code

Let's make a quick review of what's happening in the app. Open the `app.js` file and you find that these lines of code create the Azure Cosmos DB resources.

- The `DocumentClient` is initialized.

```
this.client = new DocumentClient("https://FILLME.documents.azure.com",
      "FILLME",
      new ConnectionPolicy(),
      ConsistencyLevel.Session);
```

- A new database is created.

```
Database database = new Database();
database.setId(databaseName);

this.client.createDatabase(database, null);
```

- A new collection is created.

```
DocumentCollection collectionInfo = new DocumentCollection();
collectionInfo.setId(collectionName);

// DocumentDB collections can be reserved with throughput
// specified in request units/second. 1 RU is a normalized
// request equivalent to the read of a 1KB document. Here we
// create a collection with 400 RU/s.
RequestOptions requestOptions = new RequestOptions();
requestOptions.setOfferThroughput(400);

this.client.createCollection(databaseLink, collectionInfo, requestOptions);
```

- Some documents are created.

```
// Any Java object within your code can be serialized into JSON and written to Azure Cosmos DB
Family andersenFamily = new Family();
andersenFamily.setId("Andersen.1");
andersenFamily.setLastName("Andersen");
// More properties

String collectionLink = String.format("/dbs/%s/colls/%s", databaseName, collectionName);
this.client.createDocument(collectionLink, family, new RequestOptions(), true);
```

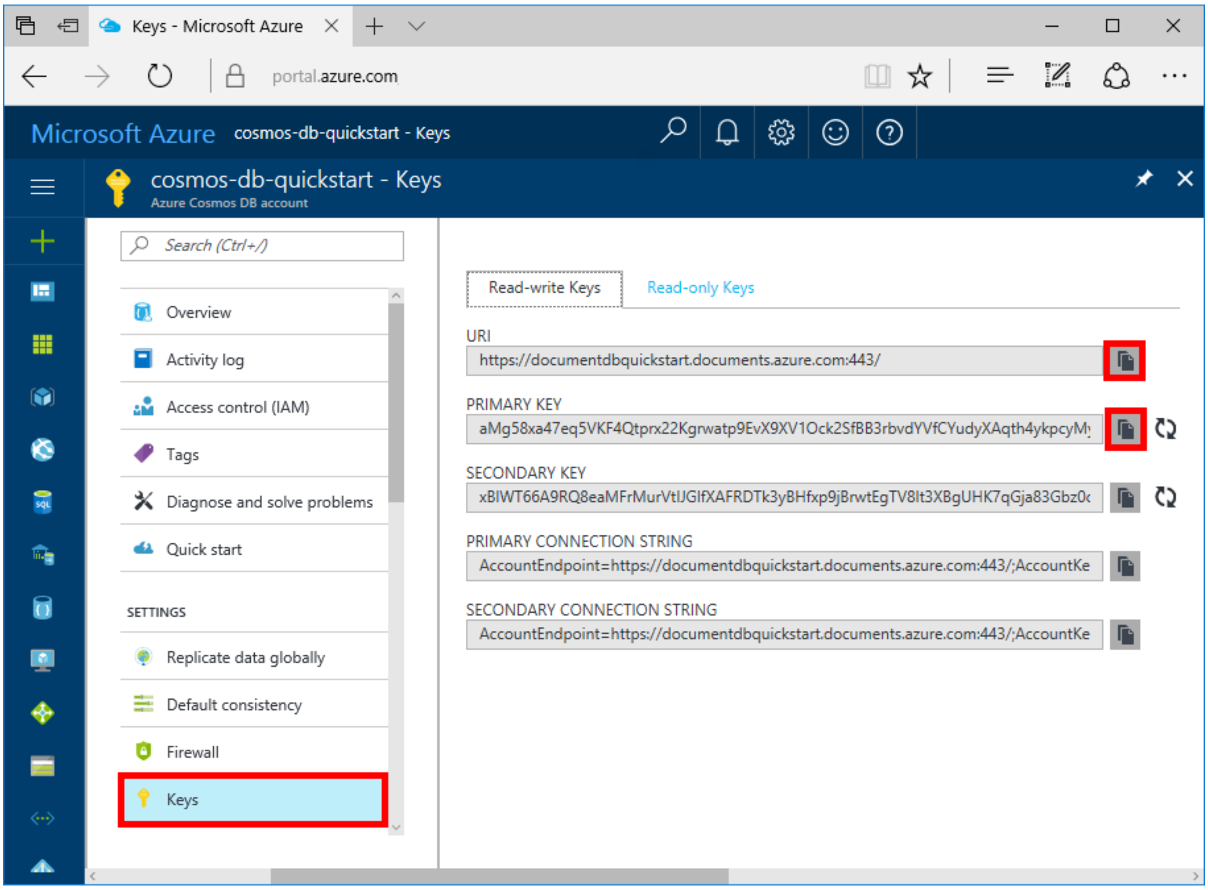
- A SQL query over JSON is performed.

```
FeedOptions queryOptions = new FeedOptions();
queryOptions.setPageSize(-1);
queryOptions.setEnableCrossPartitionQuery(true);

String collectionLink = String.format("/dbs/%s/colls/%s", databaseName, collectionName);
FeedResponse<Document> queryResults = this.client.queryDocuments(
   collectionLink,
   "SELECT * FROM Family WHERE Family.lastName = 'Andersen'", queryOptions);

System.out.println("Running SQL query...");
for (Document family : queryResults.getQueryIterable()) {
   System.out.println(String.format("\tRead %s", family));
}
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. In the Azure portal, in your Azure Cosmos DB account, in the left navigation click **Keys**, and then click **Read-write Keys**. You'll use the copy buttons on the right side of the screen to copy the URI and Primary Key into the `Program.java` file in the next step.



URI
https://documentdbquickstart.documents.azure.com:443/

PRIMARY KEY
aMg58xa47eq5VKF4Qtprx22Kgrwatp9EvX9XV1Ock2SfBB3rbvdYVfCYudyXAqth4ykpcyMy

SECONDARY KEY
xBlWT66A9RQ8eaMFrMurVtIJGlfXAFRDTk3yBHfxp9jBrwtEgTV8lt3XBgUHK7qGja83Gbz0c

PRIMARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

SECONDARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

2. In Open the `Program.java` file.

3. Copy your URI value from the portal (using the copy button) and make it the value of the endpoint to the DocumentClient constructor in `Program.java`.

   ```
   "https://FILLME.documents.azure.com"
   ```

4. Then copy your PRIMARY KEY value from the portal and make it the value of the master key to the DocumentClient constructor in `Program.java'. You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

```
config.primaryKey "FILLME"
```

Run the app

1. Run `mvn package` in a terminal to install required npm modules
2. Run `mvn exec:java -D exec.mainClass=GetStarted.Program` in a terminal to start your Java application.

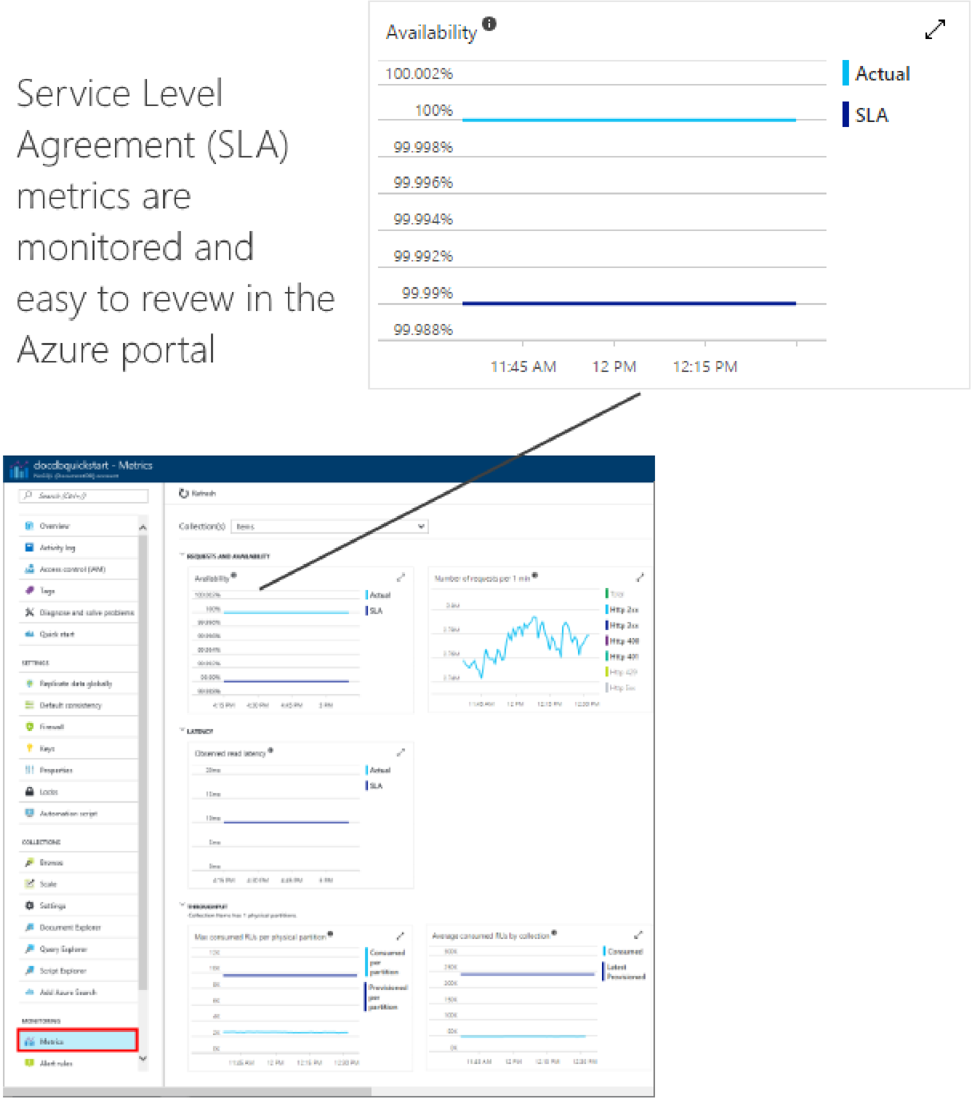
You can now go back to Data Explorer and see query, modify, and work with this new data.

Review SLAs in the Azure portal

Now that your app is up and running, you'll want to ensure business continuity and watch user access to ensure high availability. You can use the Azure portal to review the availability, latency, throughput, and consistency of your collection.

Each graph that's associated with the Azure Cosmos DB Service Level Agreements (SLAs) provides a line that shows the quota required to meet the SLA and your actual usage, giving you a clear view into your database performance. Additional metrics, such as storage usage and number of requests per minute, are also included in the portal.

- In the Azure portal, in the left pane, under **Monitoring**, click **Metrics**.



Service Level Agreement (SLA) metrics are monitored and easy to revew in the Azure portal

Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart in the Azure portal

with the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a collection using the Data Explorer, and run an app. You can now import additional data to your Cosmos DB account.

Import data into Azure Cosmos DB

Azure Cosmos DB: Build a DocumentDB API app with Node.js and the Azure portal

6/1/2017 • 6 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quick start demonstrates how to create an Azure Cosmos DB account, document database, and collection using the Azure portal. You then build and run a console app built on the DocumentDB Node.js API.

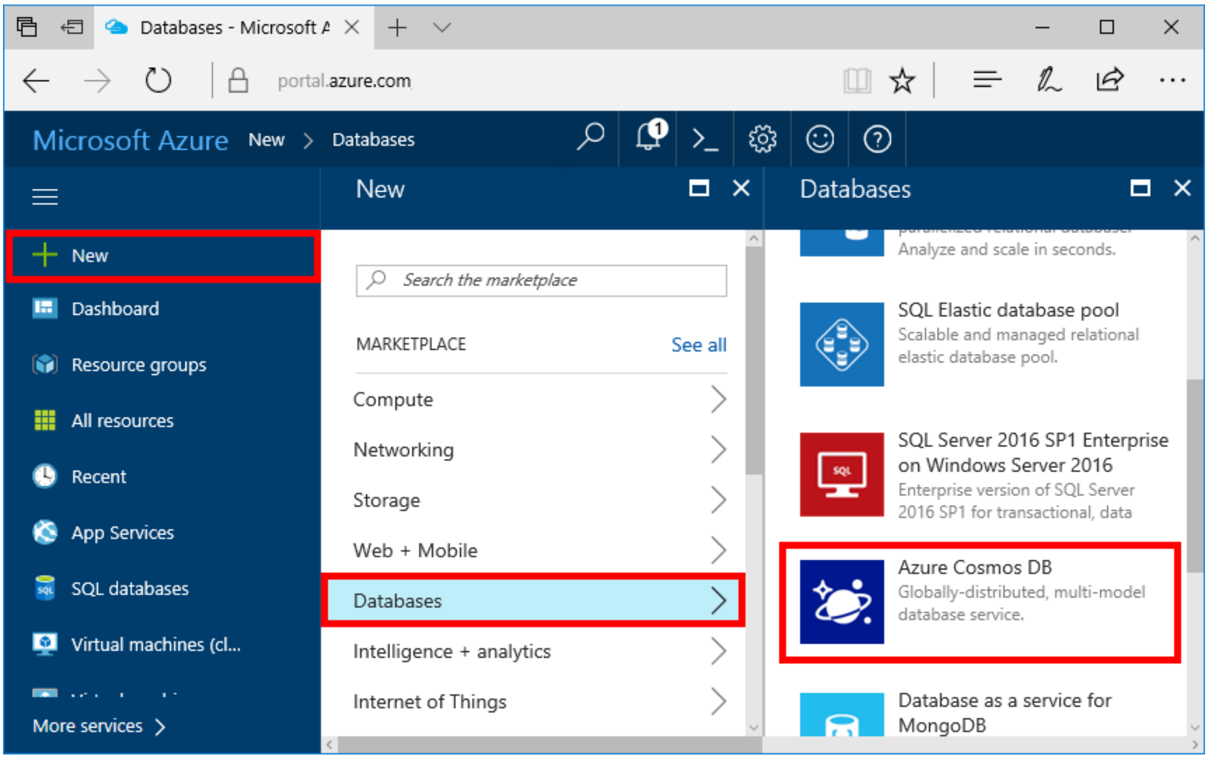
Prerequisites

- Before you can run this sample, you must have the following prerequisites:
  - Node.js version v0.10.29 or higher
  - Git

If you don't have an Azure subscription, create a free account before you begin.

Create a database account

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.



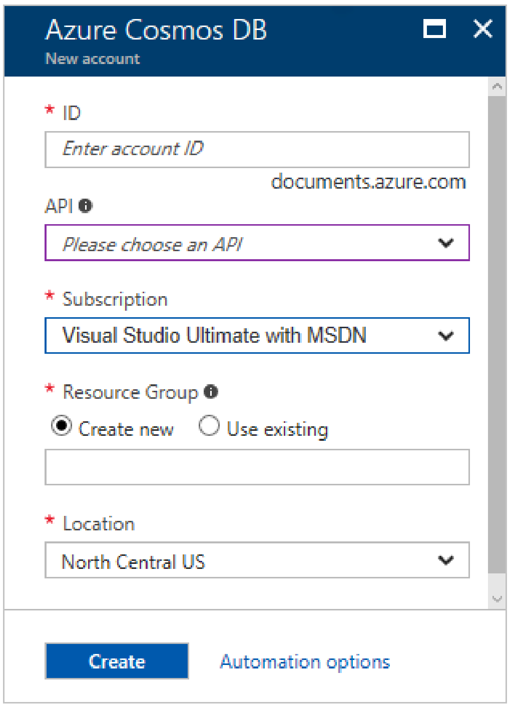
3. On the **New account** blade, specify the configuration that you want for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article we program against the DocumentDB API, so choose **SQL (DocumentDB)** as you

fill out the form. But if you have graph data for a social media app, or key/value (table) data, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

Complete the fields on the **New account** blade, using the information in the following screenshot as a guide. When you set up your account, choose unique values that do not match those in the screenshot.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

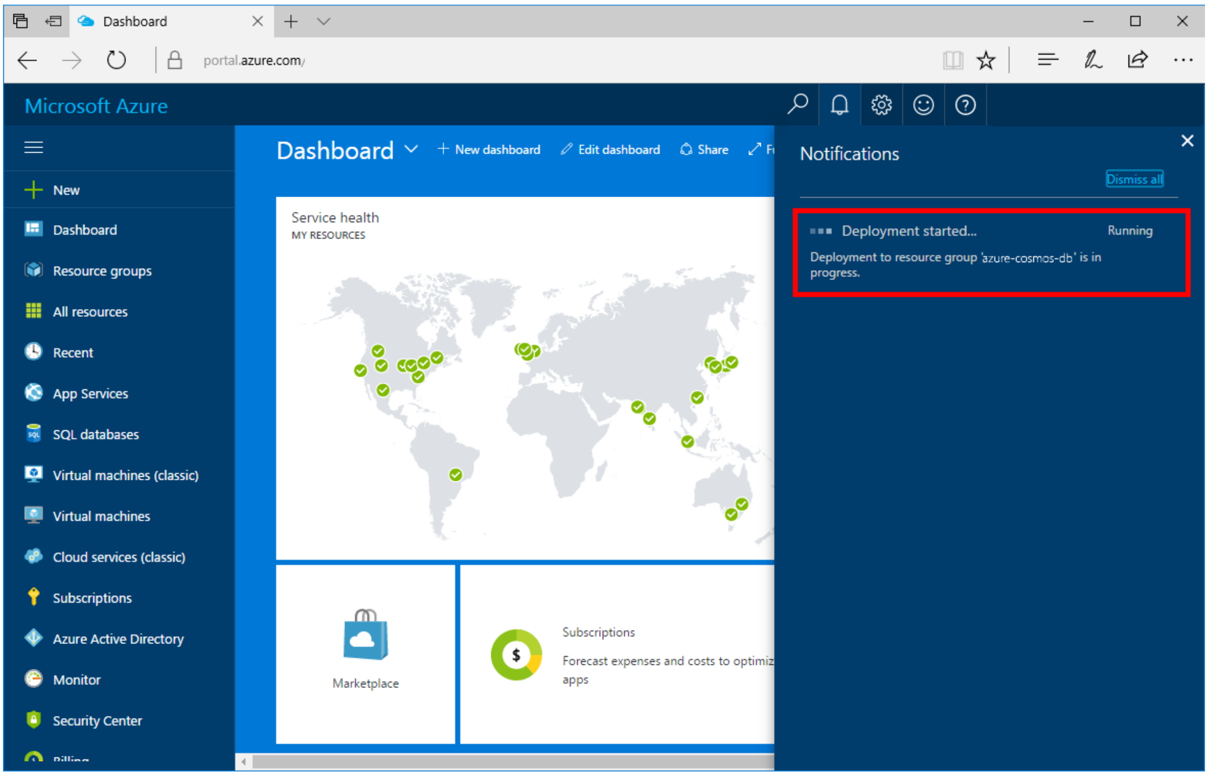
* Subscription
Visual Studio Ultimate with MSDN

* Resource Group
● Create new ○ Use existing

* Location
North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that identifies your Azure Cosmos DB account. The string *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 through 50 characters.
API	SQL (DocumentDB)	We program against the DocumentDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for your Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource-group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location that's closest to your users to give them the fastest access to the data.

4. Click **Create** to create the account.
5. On the top toolbar, click **Notifications** to monitor the deployment process.

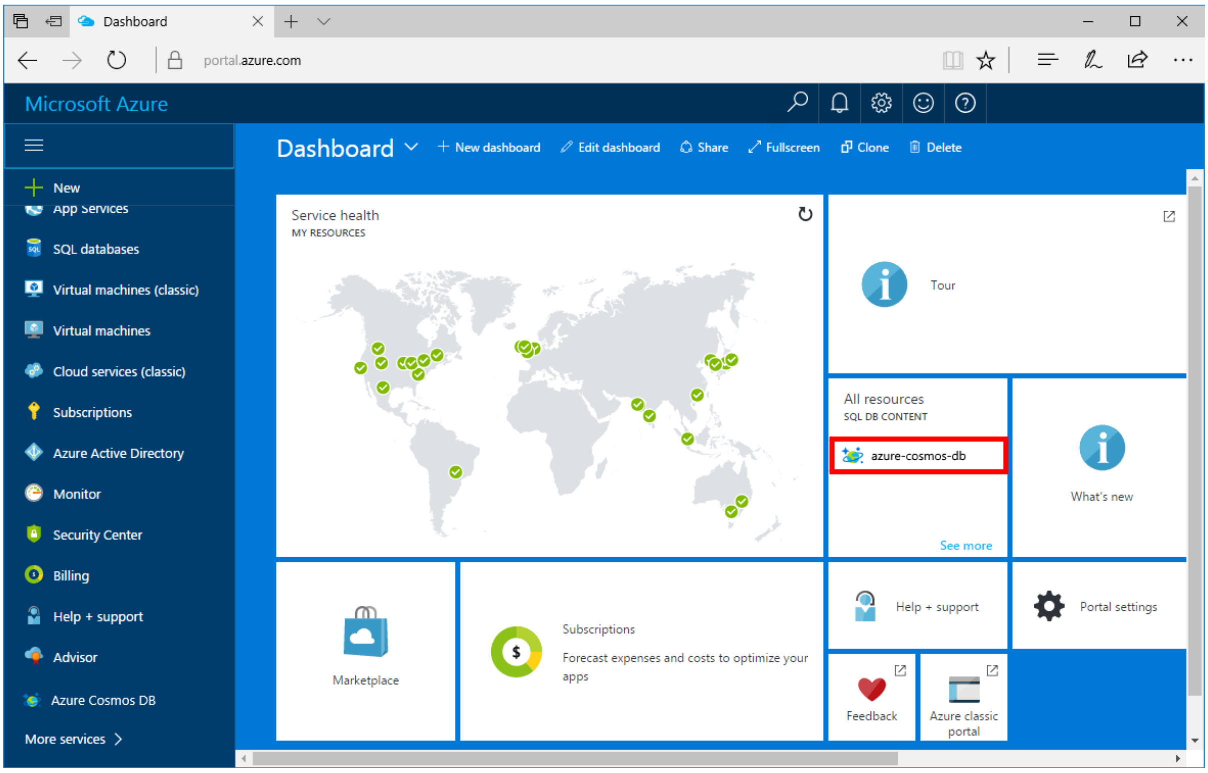


Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

Forecast expenses and costs to optimiz apps

6. When the deployment is complete, open the new account from the **All Resources** tile.



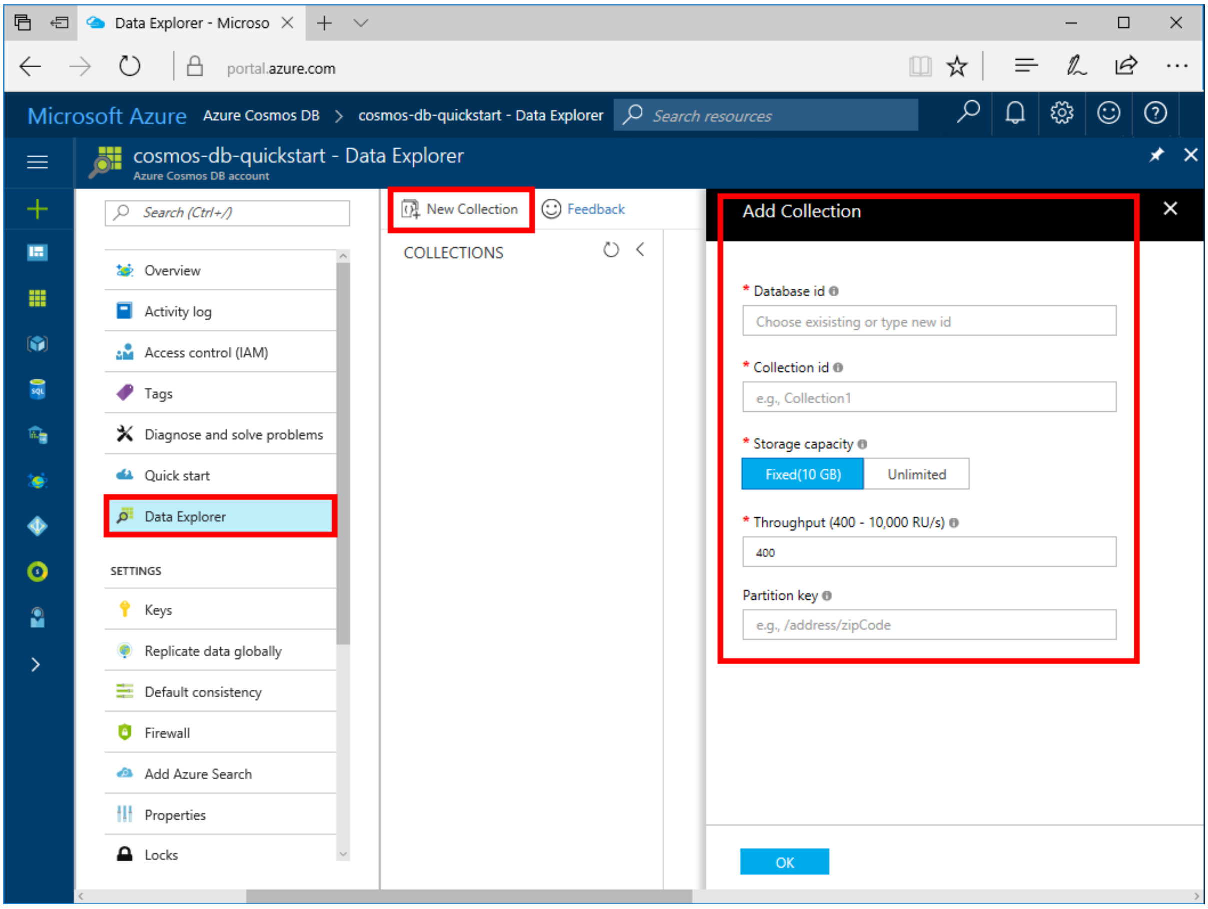
Forecast expenses and costs to optimize your apps

Add a collection

You can now use Data Explorer to create a collection and add data to your database.

1. In the Azure portal, in the left pane, click **Data Explorer**.

2. On the **Data Explorer** blade, click **New Collection**, and then provide the following information:



* Database id
Choose exisisting or type new id

* Collection id
e.g., Collection1

* Storage capacity
Fixed(10 GB) | Unlimited

* Throughput (400 - 10,000 RU/s)
400

Partition key
e.g., /address/zipCode

SETTING	SUGGESTED VALUE	DESCRIPTION
Database id	Items	The ID for your new database. Database names must contain from 1 through 255 characters, and they cannot contain /, \, #, ?, or a trailing space.
Collection id	ToDoList	The ID for your new collection. Collection names have the same character requirements as database IDs.
Storage capacity	Fixed (10 GB)	Use the default value. This is the storage capacity of the database.
Throughput	400 RU	Use the default value. If you want to reduce latency, you can scale up the throughput later.
Partition key	/userid	A partition key that distributes data evenly to each partition. Selecting the correct partition key is important in creating a performant collection. To learn more, see Designing for partitioning.

3. After you've completed the form, click **OK**.

Clone the sample application

Now let's clone a DocumentDB API app from github, set the connection string, and run it. You see how easy it is to

work with data programmatically.

1. Open a git terminal window, such as git bash, and `CD` to a working directory.

2. Run the following command to clone the sample repository.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-documentdb-nodejs-getting-started.git
```

Review the code

Let's make a quick review of what's happening in the app. Open the `app.js` file and you find that these lines of code create the Azure Cosmos DB resources.

- The `documentClient` is initialized.

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

- A new database is created.

```
client.createDatabase(config.database, (err, created) => {
   if (err) reject(err)
   else resolve(created);
});
```

- A new collection is created.

```
client.createCollection(databaseUrl, config.collection, { offerThroughput: 400 }, (err, created) => {
   if (err) reject(err)
   else resolve(created);
});
```

- Some documents are created.

```
client.createDocument(collectionUrl, document, (err, created) => {
   if (err) reject(err)
   else resolve(created);
});
```

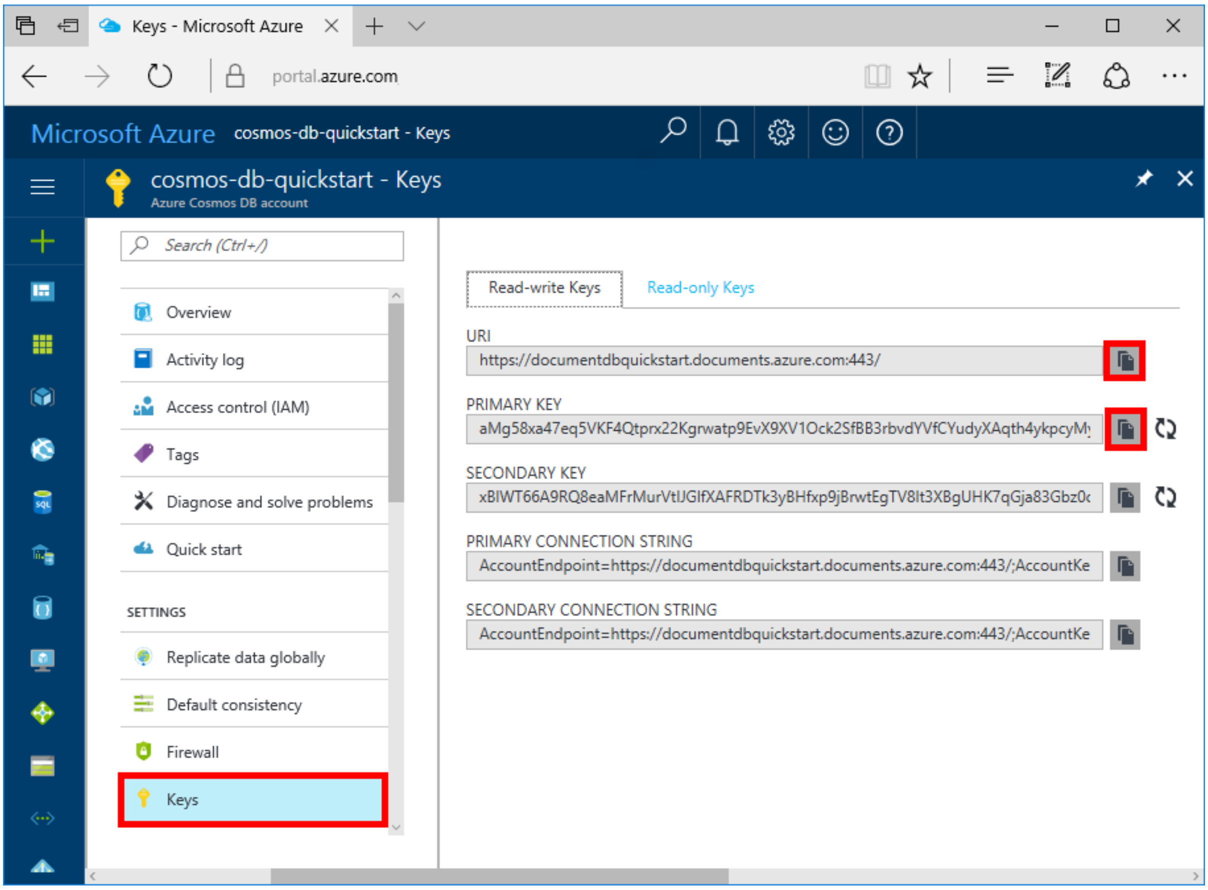
- A SQL query over JSON is performed.

```
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);
   }
});
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. In the Azure portal, in your Azure Cosmos DB account, in the left navigation click **Keys**, and then click **Read-write Keys**. You'll use the copy buttons on the right side of the screen to copy the URI and Primary Key into the `config.js` file in the next step.



URI
https://documentdbquickstart.documents.azure.com:443/

PRIMARY KEY
aMg58xa47eq5VKF4Qtprx22Kgrwatp9EvX9XV1Ock2SfBB3rbvdYVfCYudyXAqth4ykpcyMy

SECONDARY KEY
xBIWT66A9RQ8eaMFrMurVtIJGIfXAFRDTk3yBHfxp9jBrwtEgTV8It3XBgUHK7qGja83Gbz0c

PRIMARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

SECONDARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

2. In Open the `config.js` file.

3. Copy your URI value from the portal (using the copy button) and make it the value of the endpoint key in `config.js`.

```
config.endpoint = "https://FILLME.documents.azure.com"
```

4. Then copy your PRIMARY KEY value from the portal and make it the value of the `config.primaryKey` in `config.js`. You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

```
config.primaryKey "FILLME"
```

Run the app

1. Run `npm install` in a terminal to install required npm modules

2. Run `node app.js` in a terminal to start your node application.

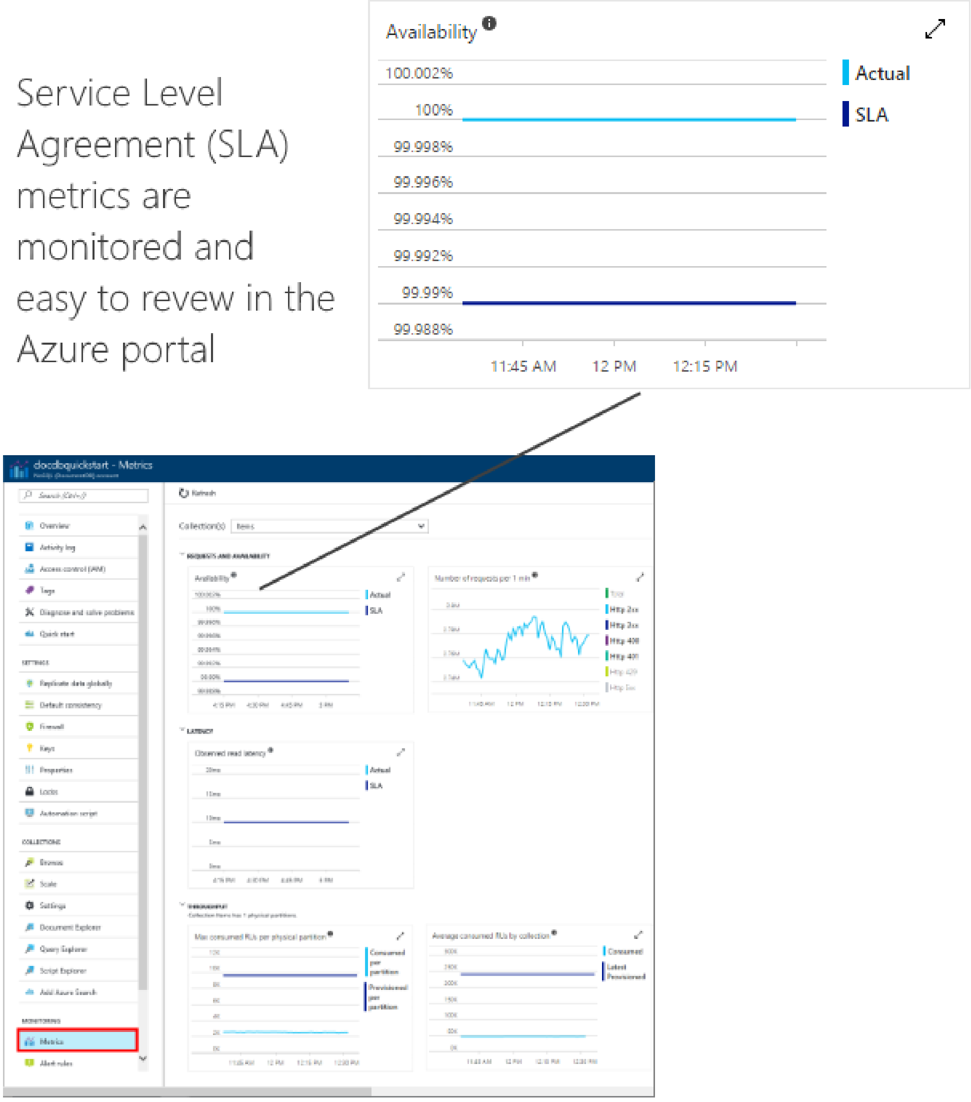
You can now go back to Data Explorer and see query, modify, and work with this new data.

Review SLAs in the Azure portal

Now that your app is up and running, you'll want to ensure business continuity and watch user access to ensure high availability. You can use the Azure portal to review the availability, latency, throughput, and consistency of your collection.

Each graph that's associated with the Azure Cosmos DB Service Level Agreements (SLAs) provides a line that shows the quota required to meet the SLA and your actual usage, giving you a clear view into your database performance. Additional metrics, such as storage usage and number of requests per minute, are also included in the portal.

- In the Azure portal, in the left pane, under **Monitoring**, click **Metrics**.



Service Level Agreement (SLA) metrics are monitored and easy to revew in the Azure portal

Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart in the Azure portal with the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a collection using the Data Explorer, and run an app. You can now import additional data to your Cosmos DB account.

Import data into Azure Cosmos DB

Azure Cosmos DB: Build a DocumentDB API app with Python and the Azure portal

6/1/2017 • 6 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quick start demonstrates how to create an Azure Cosmos DB account, document database, and collection using the Azure portal. You then build and run a console app built on the DocumentDB Python API.

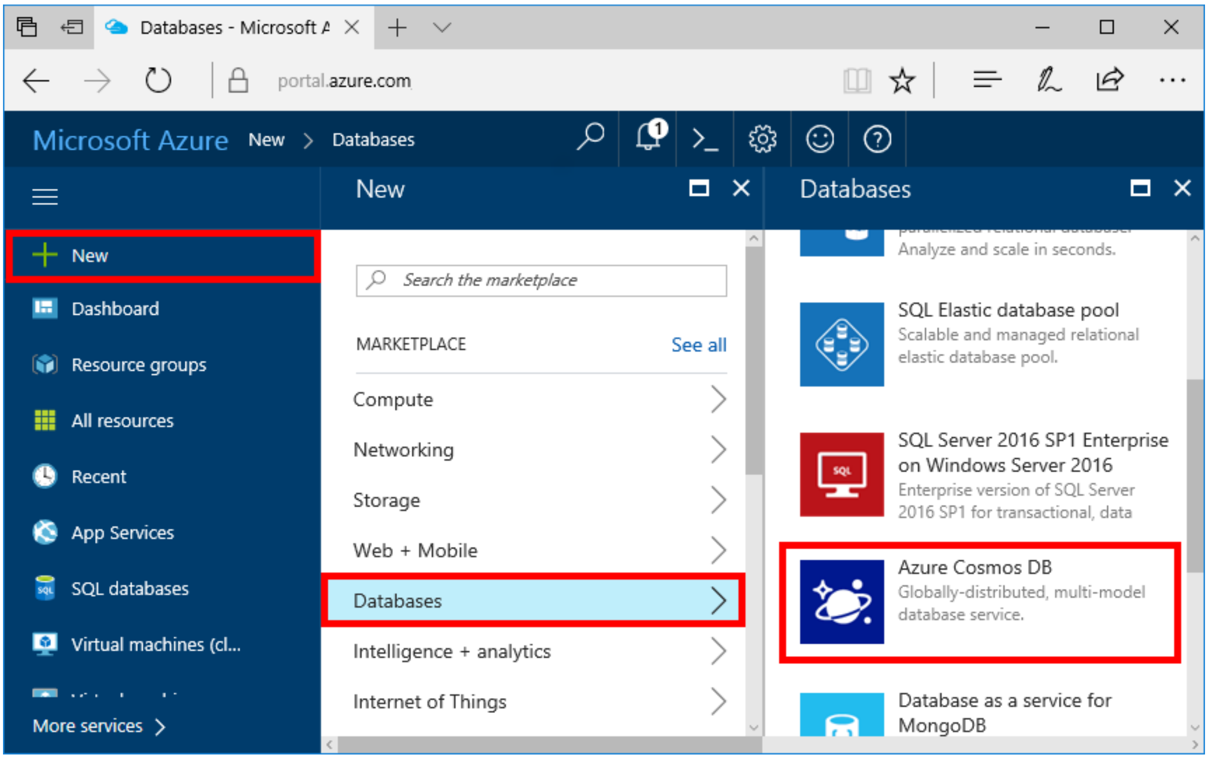
Prerequisites

- Before you can run this sample, you must have the following prerequisites:
  - Visual Studio 2015 or higher.
  - Python Tools for Visual Studio from GitHub. This tutorial uses Python Tools for VS 2015.
  - Python 2.7 from python.org

If you don't have an Azure subscription, create a free account before you begin.

Create a database account

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

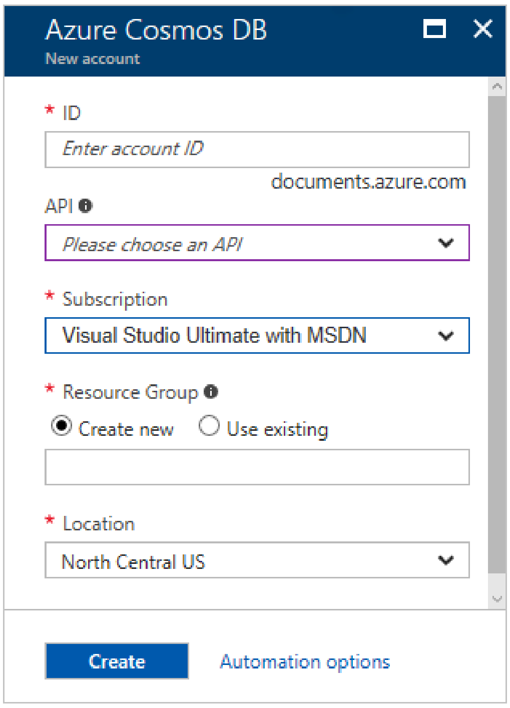


3. On the **New account** blade, specify the configuration that you want for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

In this quick-start article we program against the DocumentDB API, so choose **SQL (DocumentDB)** as you fill out the form. But if you have graph data for a social media app, or key/value (table) data, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

Complete the fields on the **New account** blade, using the information in the following screenshot as a guide. When you set up your account, choose unique values that do not match those in the screenshot.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

* Subscription
Visual Studio Ultimate with MSDN

* Resource Group
● Create new ○ Use existing

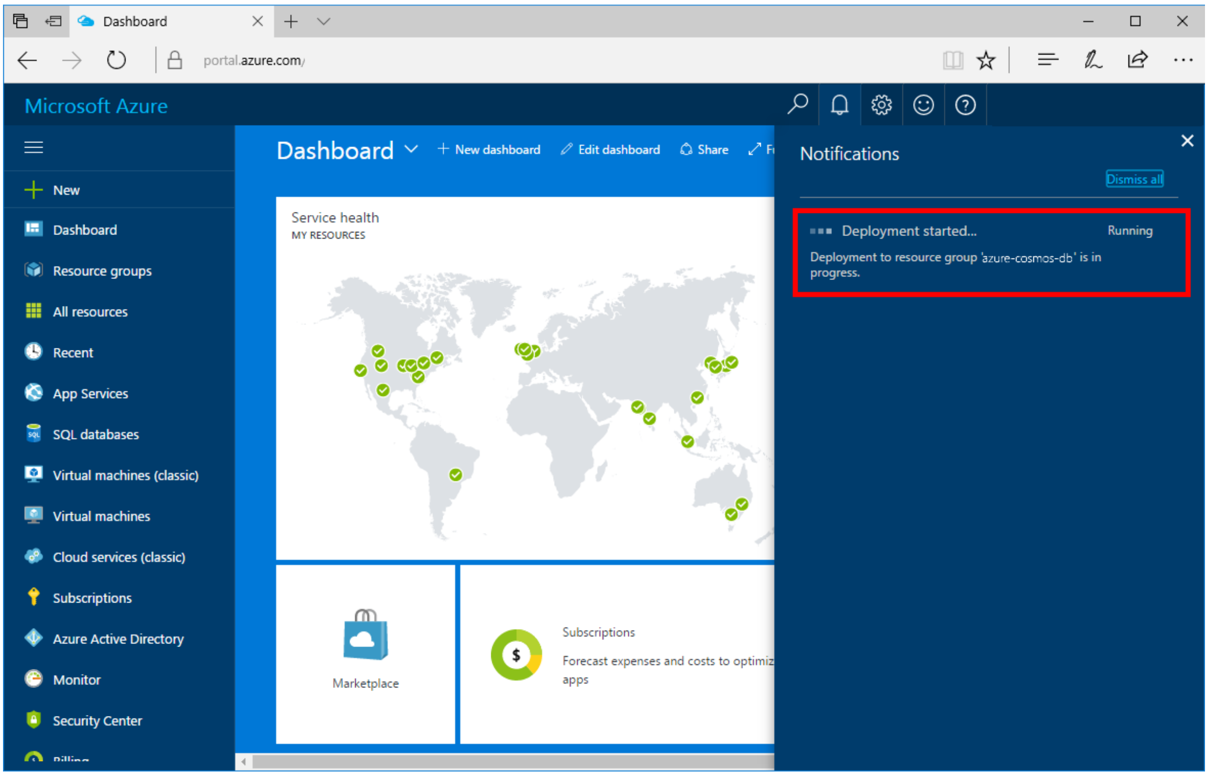
* Location
North Central US

Create Automation options

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that identifies your Azure Cosmos DB account. The string *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 through 50 characters.
API	SQL (DocumentDB)	We program against the DocumentDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for your Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource-group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location that's closest to your users to give them the fastest access to the data.

4. Click **Create** to create the account.

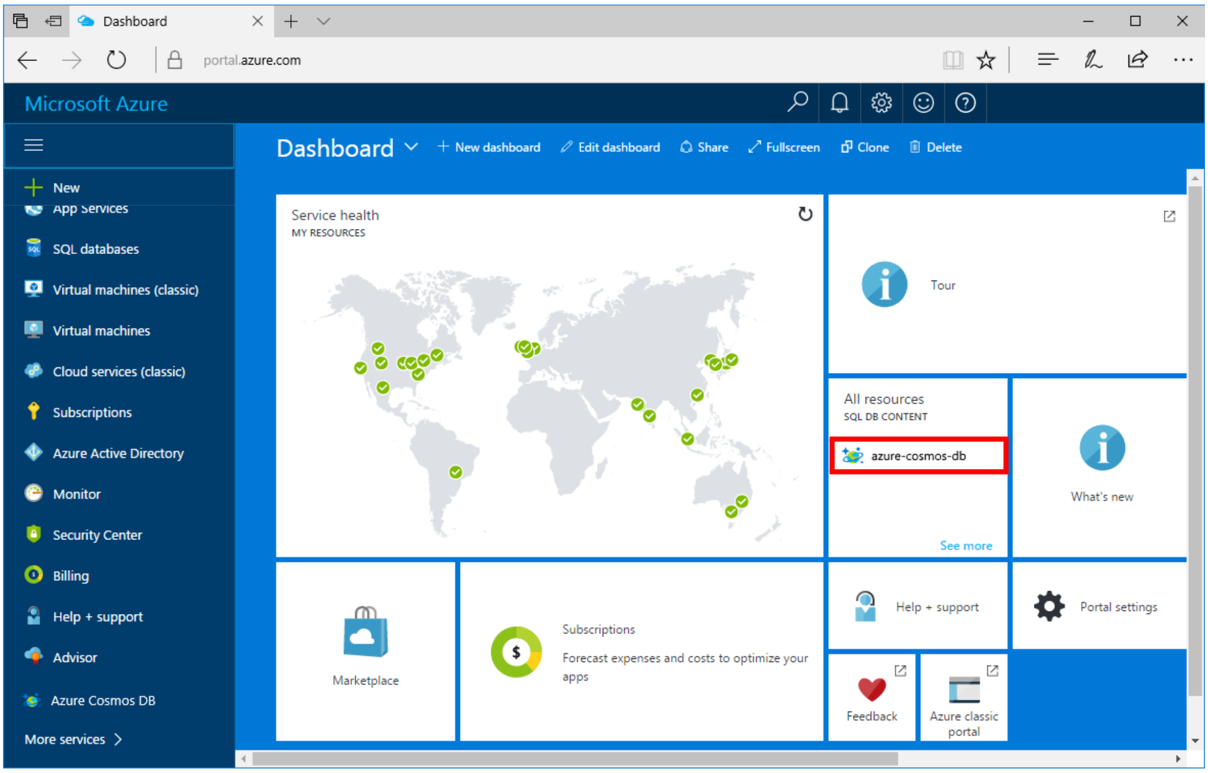
5. On the top toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the **All Resources** tile.

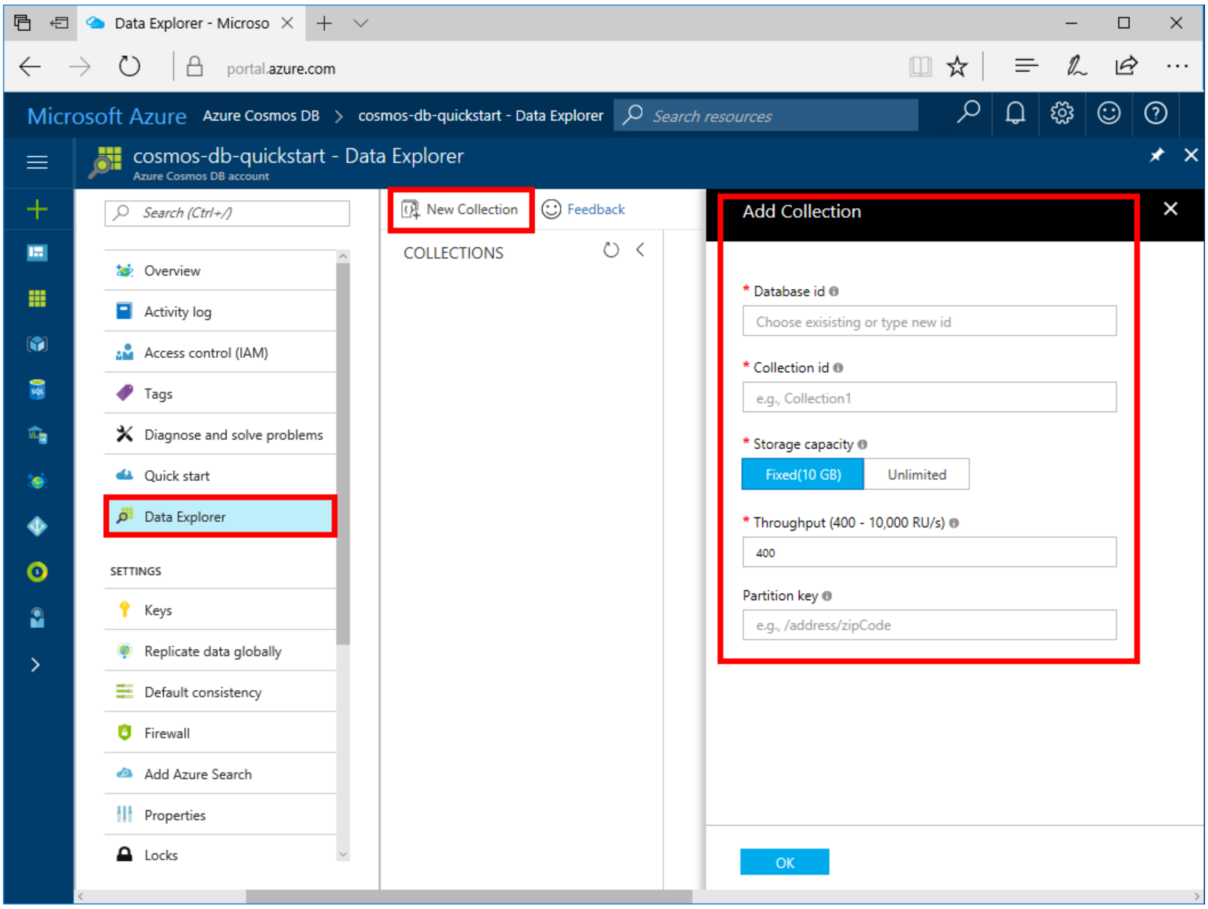


Add a collection

You can now use Data Explorer to create a collection and add data to your database.

1. In the Azure portal, in the left pane, click **Data Explorer**.

2. On the **Data Explorer** blade, click **New Collection**, and then provide the following information:



SETTING	SUGGESTED VALUE	DESCRIPTION
Database id	Items	The ID for your new database. Database names must contain from 1 through 255 characters, and they cannot contain /, \, #, ?, or a trailing space.
Collection id	ToDoList	The ID for your new collection. Collection names have the same character requirements as database IDs.
Storage capacity	Fixed (10 GB)	Use the default value. This is the storage capacity of the database.
Throughput	400 RU	Use the default value. If you want to reduce latency, you can scale up the throughput later.
Partition key	/userid	A partition key that distributes data evenly to each partition. Selecting the correct partition key is important in creating a performant collection. To learn more, see Designing for partitioning.

3. After you've completed the form, click **OK**.

Clone the sample application

Now let's clone a DocumentDB API app from github, set the connection string, and run it. You see how easy it is to

work with data programmatically.

1. Open a git terminal window, such as git bash, and `cd` to a working directory.

2. Run the following command to clone the sample repository.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-documentdb-python-getting-started.git
```

Review the code

Let's make a quick review of what's happening in the app. Open the DocumentDBGetStarted.py file and you'll find that these lines of code create the Azure Cosmos DB resources.

- The DocumentClient is initialized.

```
# Initialize the Python DocumentDB client
client = document_client.DocumentClient(config['ENDPOINT'], {'masterKey': config['MASTERKEY']})
```

- A new database is created.

```
# Create a database
db = client.CreateDatabase({ 'id': config['DOCUMENTDB_DATABASE'] })
```

- A new collection is created.

```
# Create collection options
options = {
  'offerEnableRUPerMinuteThroughput': True,
  'offerVersion': "V2",
  'offerThroughput': 400
}

# Create a collection
collection = client.CreateCollection(db['_self'], { 'id': config['DOCUMENTDB_COLLECTION'] }, options)
```

- Some documents are created.

```
# Create some documents
document1 = client.CreateDocument(collection['_self'],
  {
    'id': 'server1',
    'Web Site': 0,
    'Cloud Service': 0,
    'Virtual Machine': 0,
    'name': 'some'
  })
```

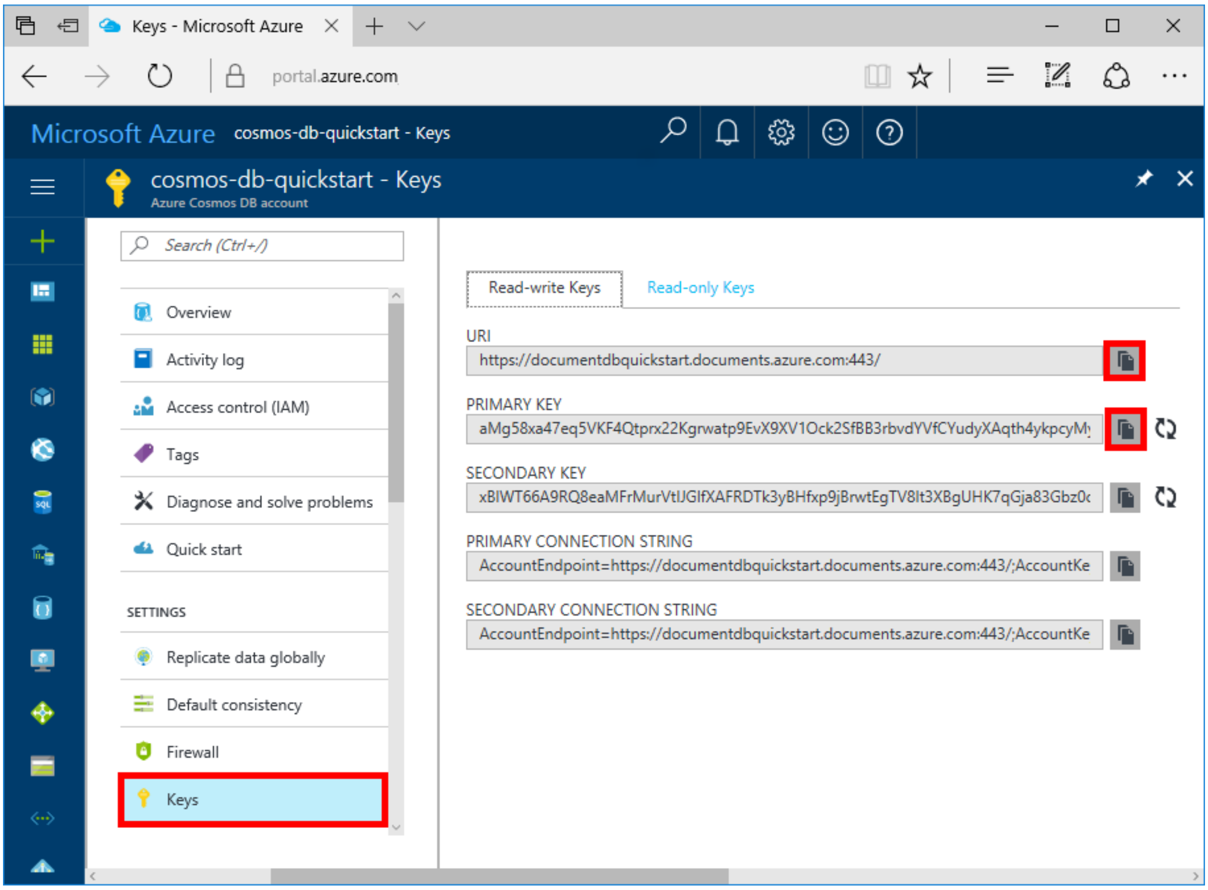
- A query is performed using SQL

```
# Query them in SQL
query = { 'query': 'SELECT * FROM server s' }

options = {}
options['enableCrossPartitionQuery'] = True
options['maxItemCount'] = 2

result_iterable = client.QueryDocuments(collection['_self'], query, options)
results = list(result_iterable);

print(results)
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. In the Azure portal, in your Azure Cosmos DB account, in the left navigation click **Keys**, and then click **Read-write Keys**. You'll use the copy buttons on the right side of the screen to copy the URI and Primary Key into the `DocumentDBGetStarted.py` file in the next step.



URI
https://documentdbquickstart.documents.azure.com:443/

PRIMARY KEY
aMg58xa47eq5VKF4Qtprx22Kgrwatp9EvX9XV1Ock2SfBB3rbvdYVfCYudyXAqth4ykpcyMy

SECONDARY KEY
xBIWT66A9RQ8eaMFrMurVtIJGIfXAFRDTk3yBHfxp9jBrwtEgTV8It3XBgUHK7qGja83Gbz0c

PRIMARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

SECONDARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

2. In Open the `DocumentDBGetStarted.py` file.

3. Copy your URI value from the portal (using the copy button) and make it the value of the endpoint key in `DocumentDBGetStarted.py`.

   ```
   config.ENDPOINT : "https://FILLME.documents.azure.com"
   ```

4. Then copy your PRIMARY KEY value from the portal and make it the value of the `config.MASTERKEY` in `DocumentDBGetStarted.py`. You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

   ```
   config.MASTERKEY : "FILLME"
   ```

Run the app

1. In Visual Studio, right-click on the project in **Solution Explorer**, select the current Python environment, then right click.

2. Select Install Python Package, then type in **pydocumentdb**

3. Run F5 to run the application. Your app displays in your browser.

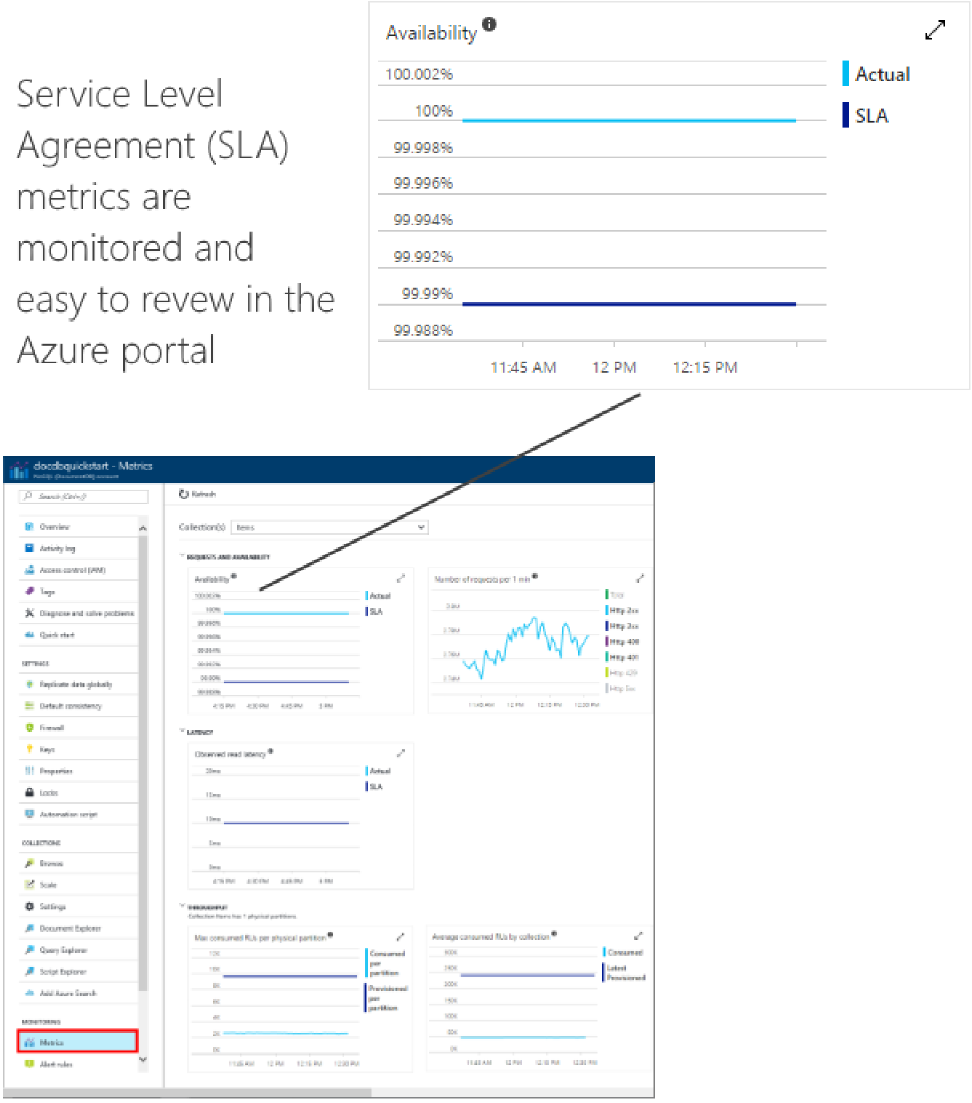
You can now go back to Data Explorer and see query, modify, and work with this new data.

Review SLAs in the Azure portal

Now that your app is up and running, you'll want to ensure business continuity and watch user access to ensure high availability. You can use the Azure portal to review the availability, latency, throughput, and consistency of your collection.

Each graph that's associated with the Azure Cosmos DB Service Level Agreements (SLAs) provides a line that shows the quota required to meet the SLA and your actual usage, giving you a clear view into your database performance. Additional metrics, such as storage usage and number of requests per minute, are also included in the portal.

- In the Azure portal, in the left pane, under **Monitoring**, click **Metrics**.



Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart in the Azure portal

with the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a collection using the Data Explorer, and run an app. You can now import additional data to your Cosmos DB account.

Import data into Azure Cosmos DB for the DocumentDB API

Azure Cosmos DB: Build a web app with .NET, Xamarin, and Facebook authentication

6/1/2017 • 7 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quick start demonstrates how to create an Azure Cosmos DB account, document database, and collection using the Azure portal. You'll then build and deploy a todo list web app built on the DocumentDB .NET API, Xamarin, and the Azure Cosmos DB authorization engine. The todo list web app implements a per-user data pattern that enables users to login using Facebook Auth and manage their own to do items.

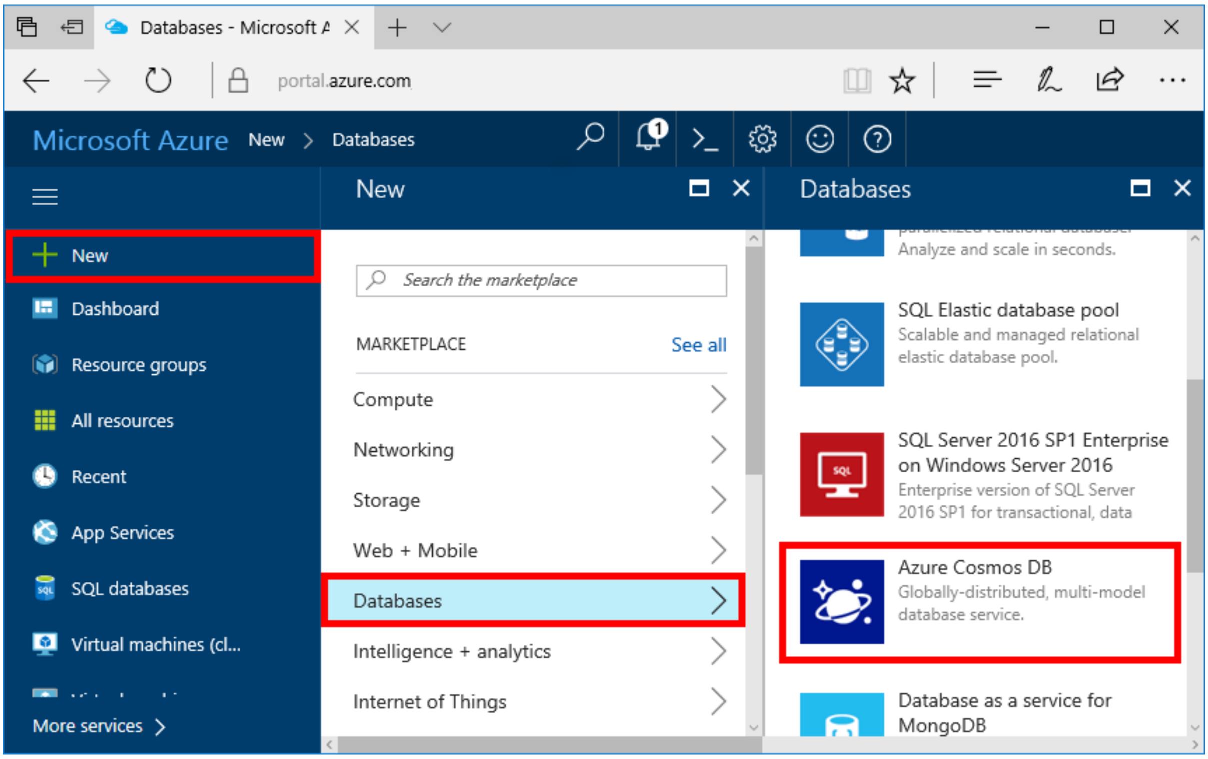
Prerequisites

If you don't already have Visual Studio 2017 installed, you can download and use the **free** Visual Studio 2017 Community Edition. Make sure that you enable **Azure development** during the Visual Studio setup.

If you don't have an Azure subscription, create a free account before you begin.

Create a database account

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.



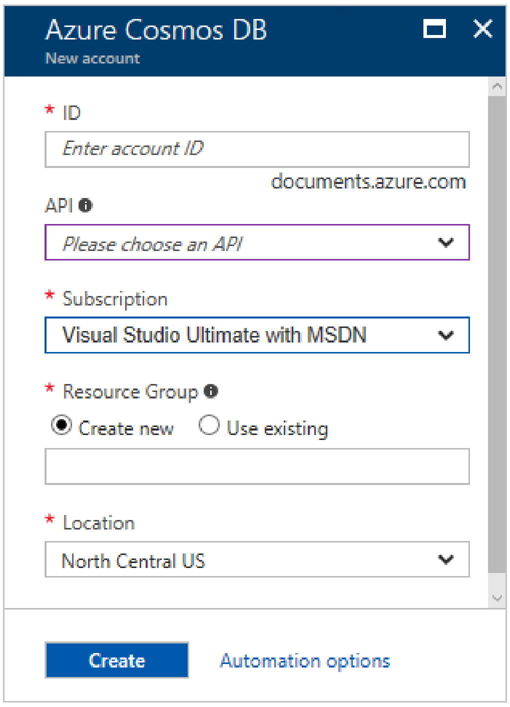
3. On the **New account** blade, specify the configuration that you want for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article we program against the DocumentDB API, so choose **SQL (DocumentDB)** as you

fill out the form. But if you have graph data for a social media app, or key/value (table) data, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

Complete the fields on the **New account** blade, using the information in the following screenshot as a guide. When you set up your account, choose unique values that do not match those in the screenshot.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

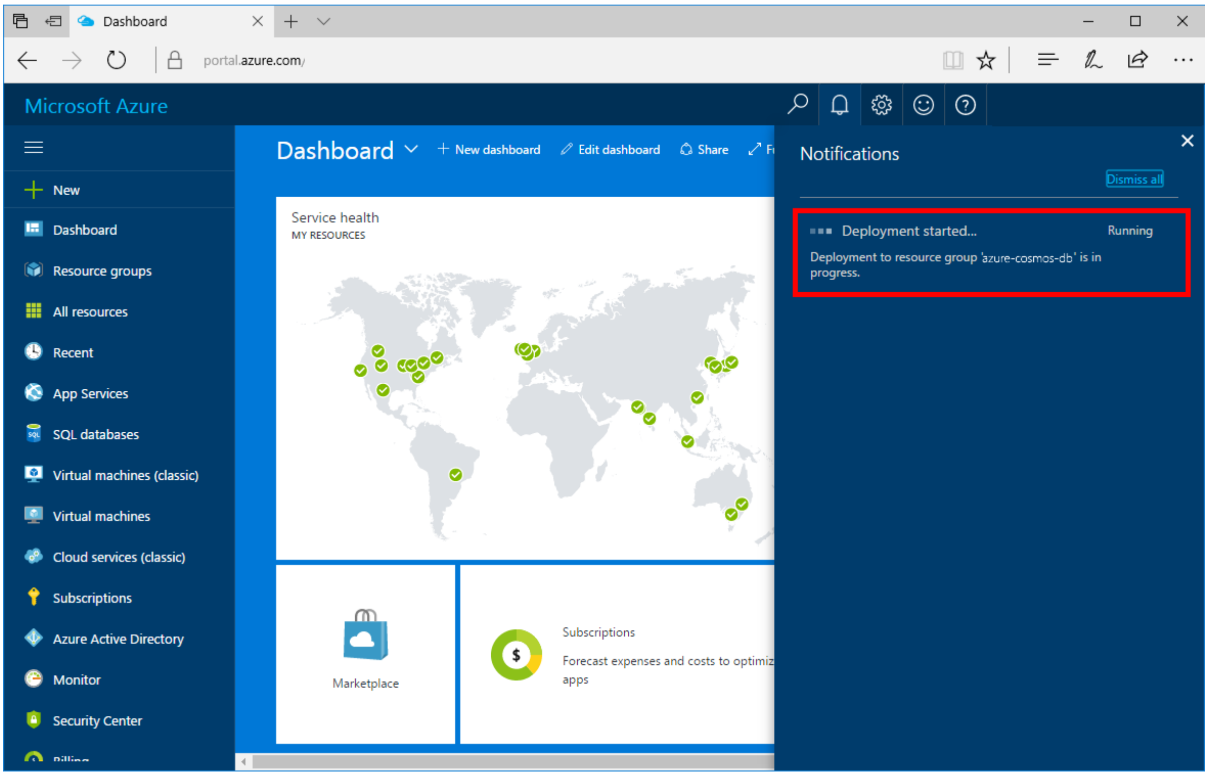
* Subscription
Visual Studio Ultimate with MSDN

* Resource Group
● Create new ○ Use existing

* Location
North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that identifies your Azure Cosmos DB account. The string *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 through 50 characters.
API	SQL (DocumentDB)	We program against the DocumentDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for your Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource-group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location that's closest to your users to give them the fastest access to the data.

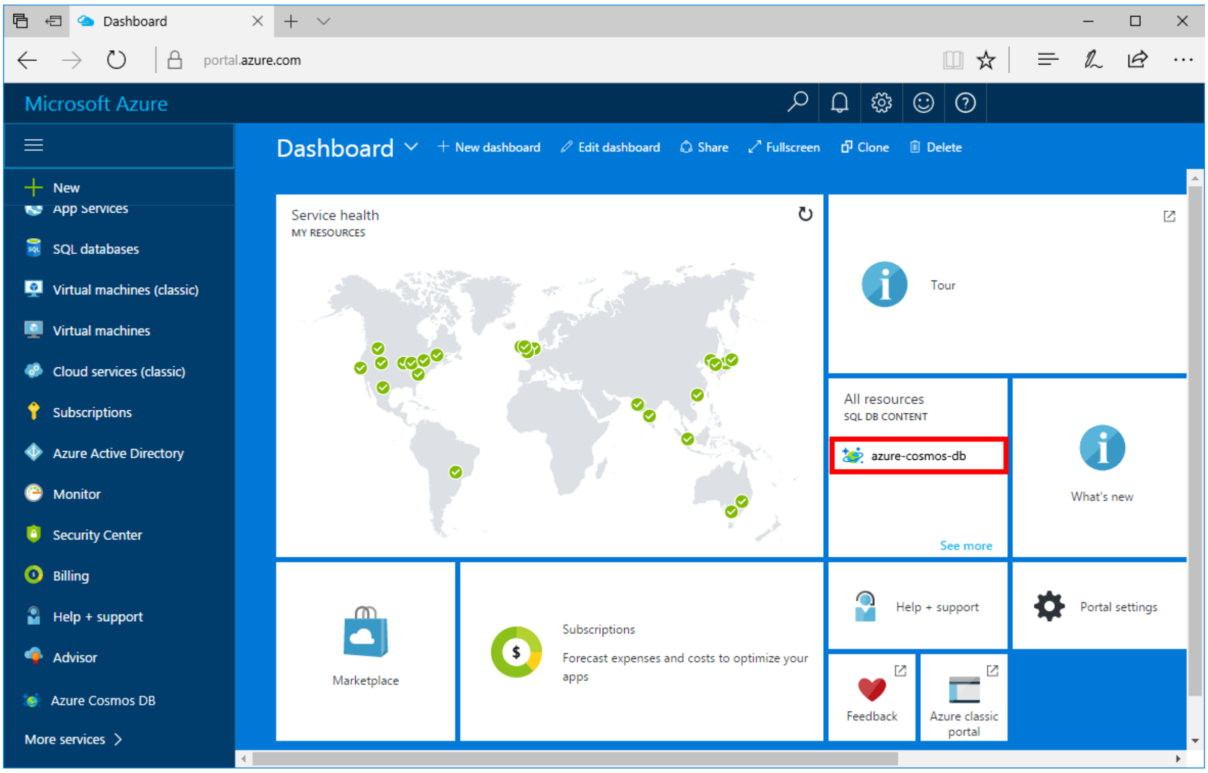
4. Click **Create** to create the account.
5. On the top toolbar, click **Notifications** to monitor the deployment process.



Deployment started...

Deployment to resource group 'azure-cosmos-db' is in progress.

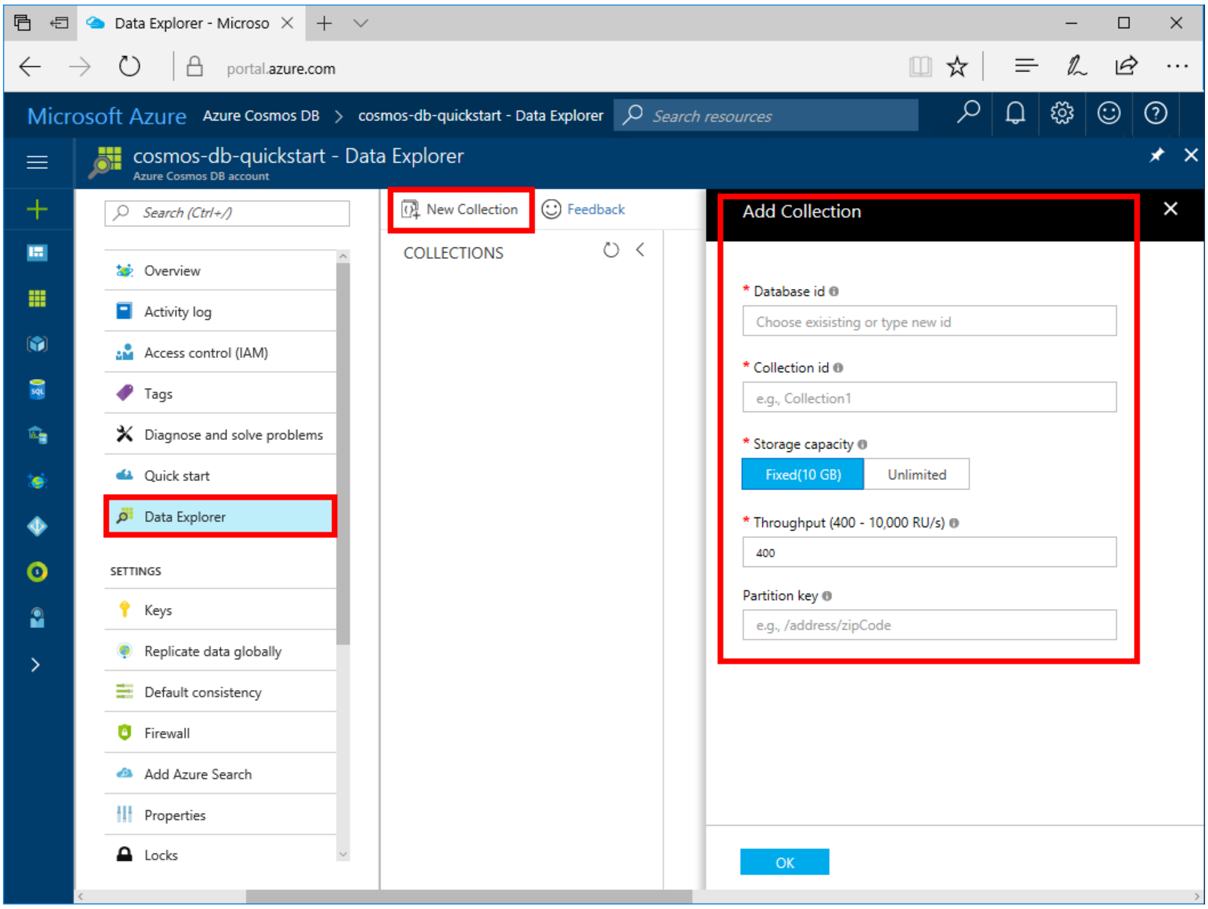
6. When the deployment is complete, open the new account from the **All Resources** tile.



Add a collection

You can now use Data Explorer to create a collection and add data to your database.

1. In the Azure portal, in the left pane, click **Data Explorer**.
2. On the **Data Explorer** blade, click **New Collection**, and then provide the following information:



Add Collection

* Database id

Choose exisisting or type new id

* Collection id

e.g., Collection1

* Storage capacity

Fixed(10 GB) | Unlimited

* Throughput (400 - 10,000 RU/s)

400

Partition key

e.g., /address/zipCode

SETTING	SUGGESTED VALUE	DESCRIPTION
Database id	Items	The ID for your new database. Database names must contain from 1 through 255 characters, and they cannot contain /, \, #, ?, or a trailing space.
Collection id	ToDoList	The ID for your new collection. Collection names have the same character requirements as database IDs.
Storage capacity	Fixed (10 GB)	Use the default value. This is the storage capacity of the database.
Throughput	400 RU	Use the default value. If you want to reduce latency, you can scale up the throughput later.
Partition key	/userid	A partition key that distributes data evenly to each partition. Selecting the correct partition key is important in creating a performant collection. To learn more, see Designing for partitioning.

3. After you've completed the form, click **OK**.

Clone the sample application

Now let's clone a DocumentDB API app from github, set the connection string, and run it. You'll see how easy it is to

work with data programmatically.

1. Open a git terminal window, such as git bash, and `cd` to a working directory.
2. Run the following command to clone the sample repository.

```
git clone https://github.com/Azure/azure-documentdb-dotnet.git
```

3. Then open the DocumentDBTodo.sln file from the samples/xamarin/UserItems/xamarin.forms folder in Visual Studio.

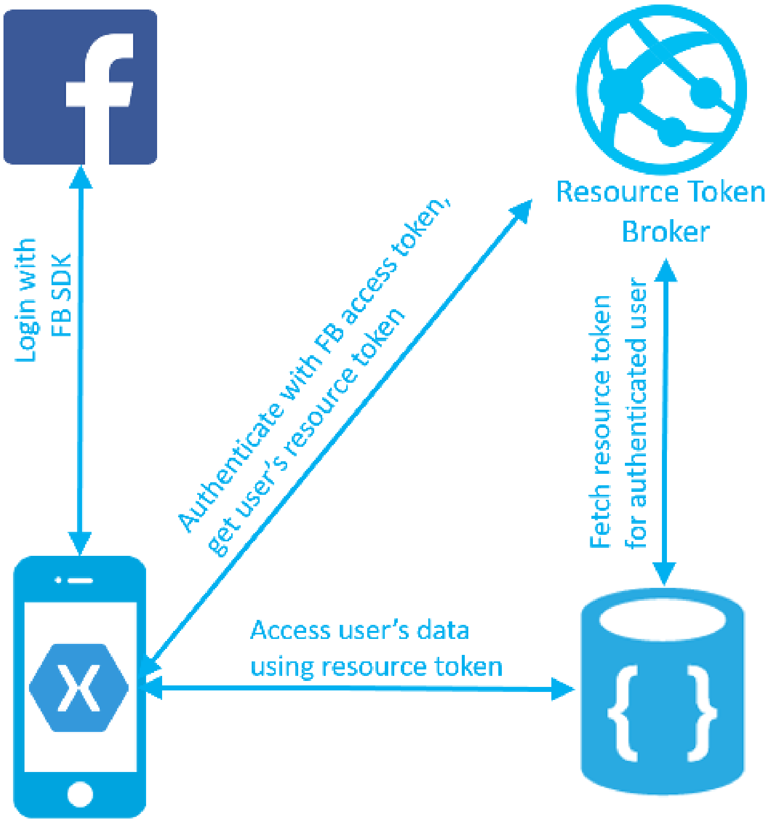
Review the code

The code in the Xamarin folder contains:

- Xamarin app. The app stores the user's todo items in a partitioned collection named UserItems.
- Resource token broker API. A simple ASP.NET Web API to broker Azure Cosmos DB resource tokens to the logged in users of the app. Resource tokens are short-lived access tokens that provide the app with the access to the logged in user's data.

The authentication and data flow is illustrated in the diagram below.

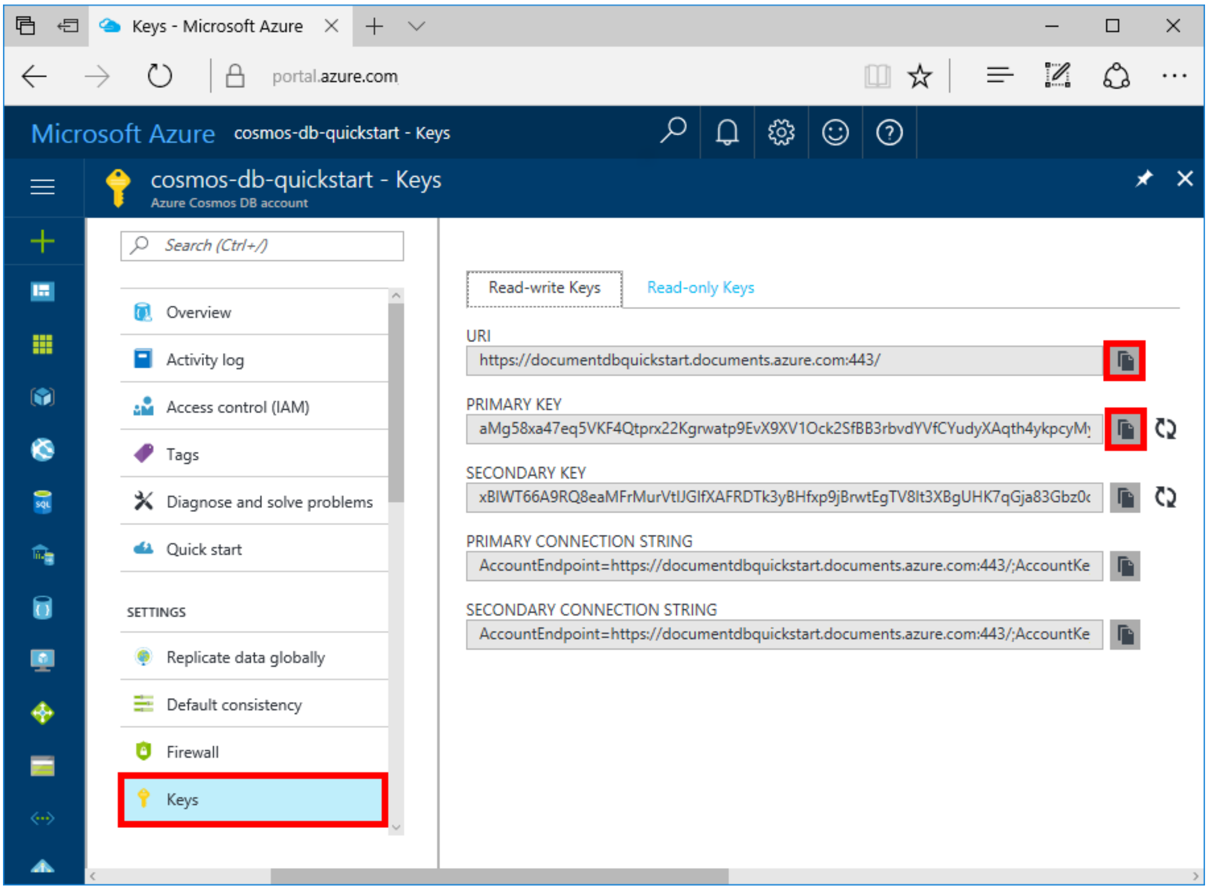
- The UserItems collection is created with the partition key '/userid'. Specifying a partition key for a collection allows Azure Cosmos DB to scale infinitely as the number of users and items grows.
- The Xamarin app allows users to login with Facebook credentials.
- The Xamarin app uses Facebook access token to authenticate with ResourceTokenBroker API
- The resource token broker API authenticates the request using App Service Auth feature, and requests an Azure Cosmos DB resource token with read/write access to all documents sharing the authenticated user's partition key.
- Resource token broker returns the resource token to the client app.
- The app accesses the user's todo items using the resource token.


Resource Token Broker
Login with FB SDK
Authenticate with FB access token, get user's resource token
Fetch resource token for authenticated user
Access user's data using resource token

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. In the Azure portal, in your Azure Cosmos DB account, in the left navigation click **Keys**, and then click **Read-write Keys**. You'll use the copy buttons on the right side of the screen to copy the URI and Primary Key into the Web.config file in the next step.



URI
https://documentdbquickstart.documents.azure.com:443/

PRIMARY KEY
aMg58xa47eq5VKF4Qtprx22Kgrwatp9EvX9XV1Ock2SfBB3rbvdYVfCYudyXAqth4ykpcyMy

SECONDARY KEY
xBIWT66A9RQ8eaMFrMurVtlJGlfXAFRDTk3yBHfxp9jBrwtEgTV8lt3XBgUHK7qGja83Gbz0c

PRIMARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

SECONDARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

2. In Visual Studio 2017, open the Web.config file in the azure-documentdb-dotnet/samples/xamarin/UserItems/ResourceTokenBroker/ResourceTokenBroker folder.

3. Copy your URI value from the portal (using the copy button) and make it the value of the accountUrl in Web.config.

```
<add key="accountUrl" value="{Azure Cosmos DB account URL}"/>
```

4. Then copy your PRIMARY KEY value from the portal and make it the value of the accountKey in Web.congif.

```
<add key="accountKey" value="{Azure Cosmos DB secret}"/>
```

You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

Build and deploy the web app

1. In the Azure portal, create an App Service website to host the Resource token broker API.
2. In the Azure portal, open the App Settings blade of the Resource token broker API website. Fill in the following app settings:

   - accountUrl - The Azure Cosmos DB account URL from the Keys tab of your Azure Cosmos DB account.
   - accountKey - The Azure Cosmos DB account master key from the Keys tab of your Azure Cosmos DB account.
   - databaseId and collectionId of your created database and collection

3. Publish the ResourceTokenBroker solution to your created website.

4. Open the Xamarin project, and navigate to TodoItemManager.cs. Fill in the values for accountURL, collectionId, databaseId, as well as resourceTokenBrokerURL as the base https url for the resource token broker website.

5. Complete the How to configure your App Service application to use Facebook login tutorial to setup Facebook authentication and configure the ResourceTokenBroker website.

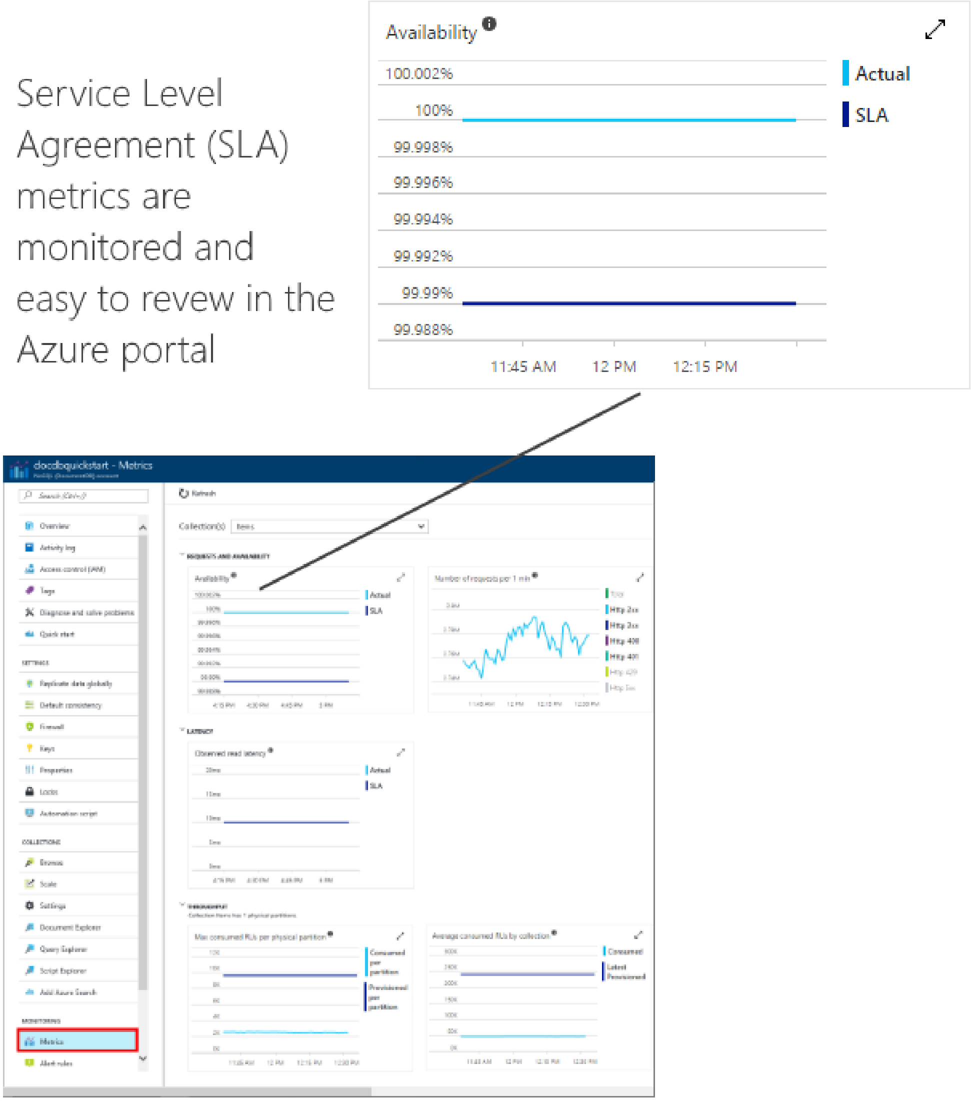
   Run the Xamarin app.

Review SLAs in the Azure portal

Now that your app is up and running, you'll want to ensure business continuity and watch user access to ensure high availability. You can use the Azure portal to review the availability, latency, throughput, and consistency of your collection.

Each graph that's associated with the Azure Cosmos DB Service Level Agreements (SLAs) provides a line that shows the quota required to meet the SLA and your actual usage, giving you a clear view into your database performance. Additional metrics, such as storage usage and number of requests per minute, are also included in the portal.

- In the Azure portal, in the left pane, under **Monitoring**, click **Metrics**.



Service Level Agreement (SLA) metrics are monitored and easy to revew in the Azure portal

Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart in the Azure portal

with the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you just created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a collection using the Data Explorer, and build and deploy a Xamarin app. You can now import additional data to your Cosmos DB account.

Import data into Azure Cosmos DB

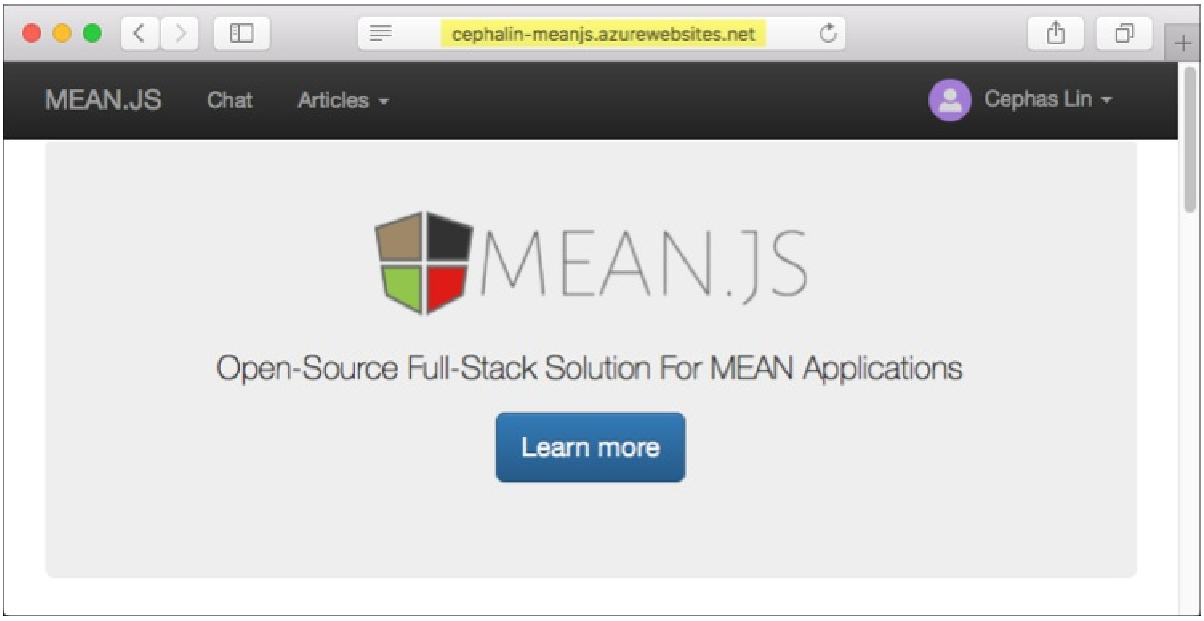
Azure Cosmos DB: Migrate an existing Node.js MongoDB web app

6/6/2017 • 6 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quickstart demonstrates how to use an existing MongoDB app written in Node.js and connect it to your Azure Cosmos DB database, which supports MongoDB client connections. In other words, your Node.js application only knows that it's connecting to a database using MongoDB APIs. It is transparent to the application that the data is stored in Azure Cosmos DB.

When you are done, you will have a MEAN application (MongoDB, Express, AngularJS, and Node.js) running on Azure Cosmos DB.


cephalin-meanjs.azurewebsites.net
MEAN.JS
Chat
Articles
Cephas Lin
MEAN.JS
Open-Source Full-Stack Solution For MEAN Applications
Learn more

Prerequisites

This quickstart requires Azure CLI 2.0. You may use the Azure Cloud Shell in the browser, or Install Azure CLI 2.0 on your own computer to run the code blocks in this tutorial.

The Azure Cloud Shell (in public preview) is a web-based shell that is preconfigured to simplify using Azure tools. With Cloud Shell, you always have the most up-to-date version of the tools available and you don't have to install, update or separately log in. Click the **Try It** button at the top right of an Azure CLI code block to launch the Cloud Shell. Then, use the **Copy** button to copy and paste the sample code into the Cloud Shell.

You can also open the Cloud Shell from the Azure portal by clicking the >_ button on the menu in the upper-right of the portal.

In addition to Azure CLI, you need Node.js and Git. You will run `az`, `npm`, and `git` commands.

You should have working knowledge of Node.js. This quickstart is not intended to help you with developing Node.js applications in general.

Clone the sample application

Open a git terminal window, such as git bash, and `cd` to a working directory.

Run the following commands to clone the sample repository. This sample repository contains the default MEAN.js application.

```
git clone https://github.com/prashanthmadi/mean
```

Run the application

Install the required packages and start the application.

```
cd mean
npm install
npm start
```

Log in to Azure

If you are using the Azure Cloud Shell, click **Try It** in the code block below, follow the on-screen prompts to log in, then proceed to the next command. If you are using an installed Azure CLI, log in to your Azure subscription with the az login command and follow the on-screen directions.

```
az login
```

Add the Azure Cosmos DB module

If you are using an installed Azure CLI, check to see if the `cosmosdb` component is already installed by running the `az` command. If `cosmosdb` is in the list of base commands, proceed to the next command.

If `cosmosdb` is not in the list of base commands, reinstall Azure CLI 2.0.

Create a resource group

Create a resource group with the az group create. An Azure resource group is a logical container into which Azure resources like web apps, databases and storage accounts are deployed and managed.

The following example creates a resource group in the West Europe region. Choose a unique name for the resource group.

```
az group create --name myResourceGroup --location "West Europe"
```

Create an Azure Cosmos DB account

Create an Azure Cosmos DB account with the az cosmosdb create command.

In the following command, please substitute your own unique Azure Cosmos DB account name where you see the `<cosmosdb_name>` placeholder. This unique name will be used as part of your Azure Cosmos DB endpoint (`https://<cosmosdb_name>.documents.azure.com/`), so the name needs to be unique across all Azure Cosmos DB accounts in Azure.

```
az cosmosdb create --name <cosmosdb_name> --resource-group myResourceGroup --kind MongoDB
```

The `--kind MongoDB` parameter enables MongoDB client connections.

When the Azure Cosmos DB account is created, the Azure CLI shows information similar to the following example.

```
{
  "databaseAccountOfferType": "Standard",
  "documentEndpoint": "https://<cosmosdb_name>.documents.azure.com:443/",
  "id": "/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/myResourceGroup/providers/Microsoft.Document
DB/databaseAccounts/<cosmosdb_name>",
  "kind": "MongoDB",
  "location": "West Europe",
  "name": "<cosmosdb_name>",
  "readLocations": [
    {
      "documentEndpoint": "https://<cosmosdb_name>-westeurope.documents.azure.com:443/",
      "failoverPriority": 0,
      "id": "<cosmosdb_name>-westeurope",
      "locationName": "West Europe",
      "provisioningState": "Succeeded"
    }
  ],
  "resourceGroup": "myResourceGroup",
  "type": "Microsoft.DocumentDB/databaseAccounts",
  "writeLocations": [
    {
      "documentEndpoint": "https://<cosmosdb_name>-westeurope.documents.azure.com:443/",
      "failoverPriority": 0,
      "id": "<cosmosdb_name>-westeurope",
      "locationName": "West Europe",
      "provisioningState": "Succeeded"
    }
  ]
}
```

Connect your Node.js application to the database

In this step, you connect your MEAN.js sample application to an Azure Cosmos DB database you just created, using a MongoDB connection string.

Retrieve the key

In order to connect to an Azure Cosmos DB database, you need the database key. Use the az cosmosdb list-keys command to retrieve the primary key.

```
az cosmosdb list-keys --name <cosmosdb_name> --resource-group myResourceGroup
```

The Azure CLI outputs information similar to the following example.

```
{
  "primaryMasterKey": "RUayjYjixJDWG5xTqIiXjC...",
  "primaryReadonlyMasterKey": "...",
  "secondaryMasterKey": "...",
  "secondaryReadonlyMasterKey": "..."
}
```

Copy the value of `primaryMasterKey` to a text editor. You need this information in the next step.

Configure the connection string in your Node.js application

In your MEAN.js repository, open `config/env/local-development.js`.

Replace the content of this file with the following code. Be sure to also replace the two `<cosmosdb_name>` placeholders with your Azure Cosmos DB account name, and the `<primary_master_key>` placeholder with the key you copied in the previous step.

```
'use strict';

module.exports = {
  db: {
    uri: 'mongodb://<cosmosdb_name>:<primary_master_key>@<cosmosdb_name>.documents.azure.com:10250/mean-dev?
ssl=true&sslverifycertificate=false'
  }
};
```

> NOTE
>
> The `ssl=true` option is important because Azure Cosmos DB requires SSL.

Save your changes.

Run the application again.

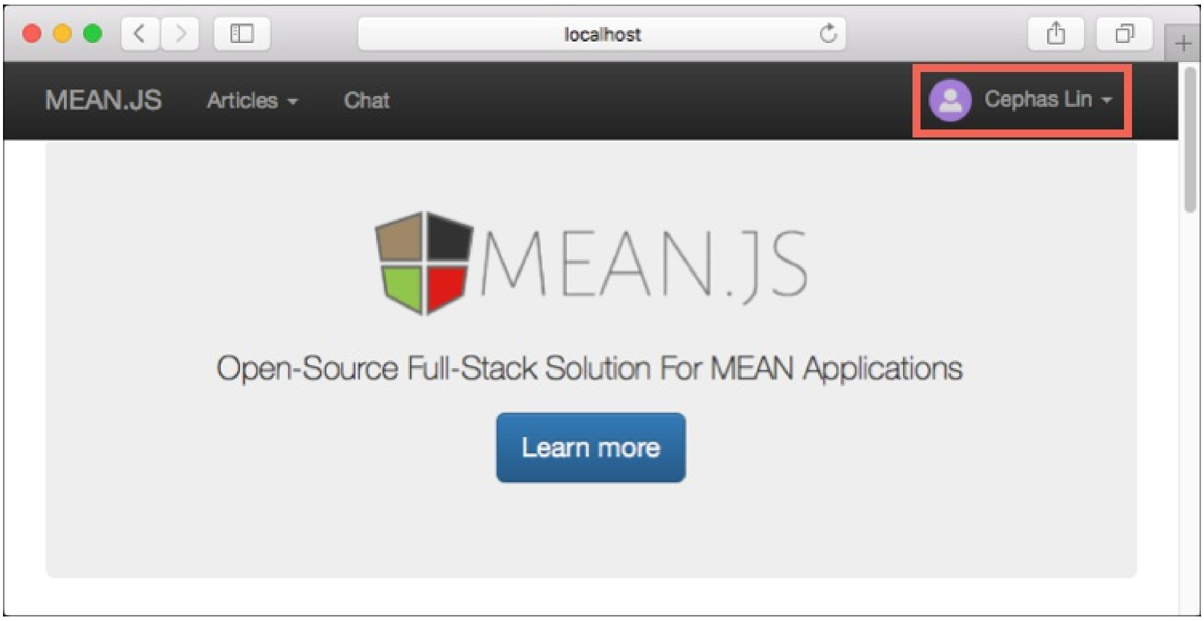
Run `npm start` again.

```
npm start
```

A console message should now tell you that the development environment is up and running.

Navigate to `http://localhost:3000` in a browser. Click **Sign Up** in the top menu and try to create two dummy users.

The MEAN.js sample application stores user data in the database. If you are successful and MEAN.js automatically signs into the created user, then your Azure Cosmos DB connection is working.

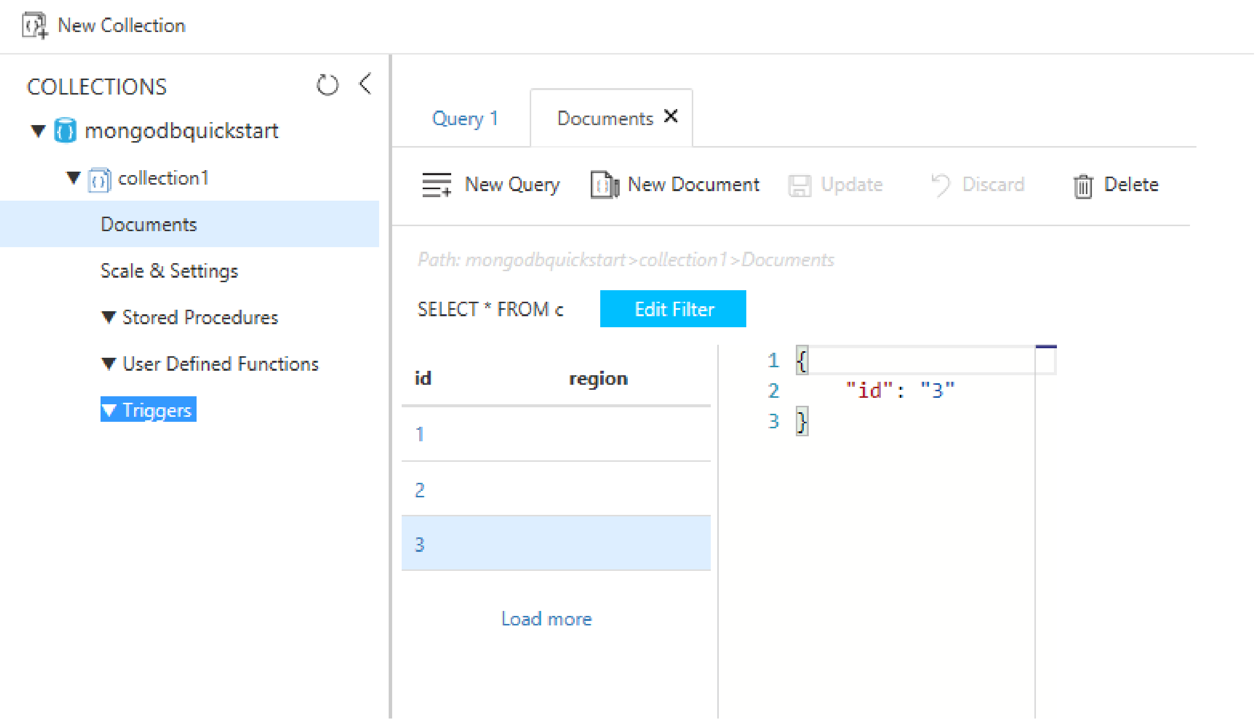


View data in Data Explorer

Data stored by an Azure Cosmos DB is available to view, query, and run business-logic on in the Azure portal.

To view, query, and work with the user data created in the previous step, login to the Azure portal in your web browser.

In the top Search box, type Azure Cosmos DB. When your Cosmos DB account blade opens, select your Cosmos DB account. In the left navigation, click Data Explorer. Expand your collection in the Collections pane, and then you can view the documents in the collection, query the data, and even create and run stored procedures, triggers, and UDFs.



Path: mongodbquickstart>collection1>Documents

```
SELECT * FROM c
```

id	region
1	
2	
3	

```
{
    "id": "3"
}
```

Deploy the Node.js application to Azure

In this step, you deploy your MongoDB-connected Node.js application to Azure Cosmos DB.

You may have noticed that the configuration file that you changed earlier is for the development environment (`/config/env/local-development.js`). When you deploy your application to App Service, it will run in the production environment by default. So now, you need to make the same change to the respective configuration file.

In your MEAN.js repository, open `config/env/production.js` .

In the `db` object, replace the value of `uri` as show in the following example. Be sure to replace the placeholders as before.

```
'mongodb://<cosmosdb_name>:<primary_master_key>@<cosmosdb_name>.documents.azure.com:10250/mean?
ssl=true&sslverifycertificate=false',
```

In the terminal, commit all your changes into Git. You can copy both commands to run them together.

```
git add .
git commit -m "configured MongoDB connection string"
```

Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart in the Azure portal with the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource

you created.

2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account and create a MongoDB collection using the Data Explorer. You can now migrate your MongoDB data to Azure Cosmos DB.

Import MongoDB data into Azure Cosmos DB

Azure Cosmos DB: Build a MongoDB API web app with .NET and the Azure portal

6/1/2017 • 5 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quick start demonstrates how to create an Azure Cosmos DB account, document database, and collection using the Azure portal. You'll then build and deploy a tasks list web app built on the MongoDB .NET driver.

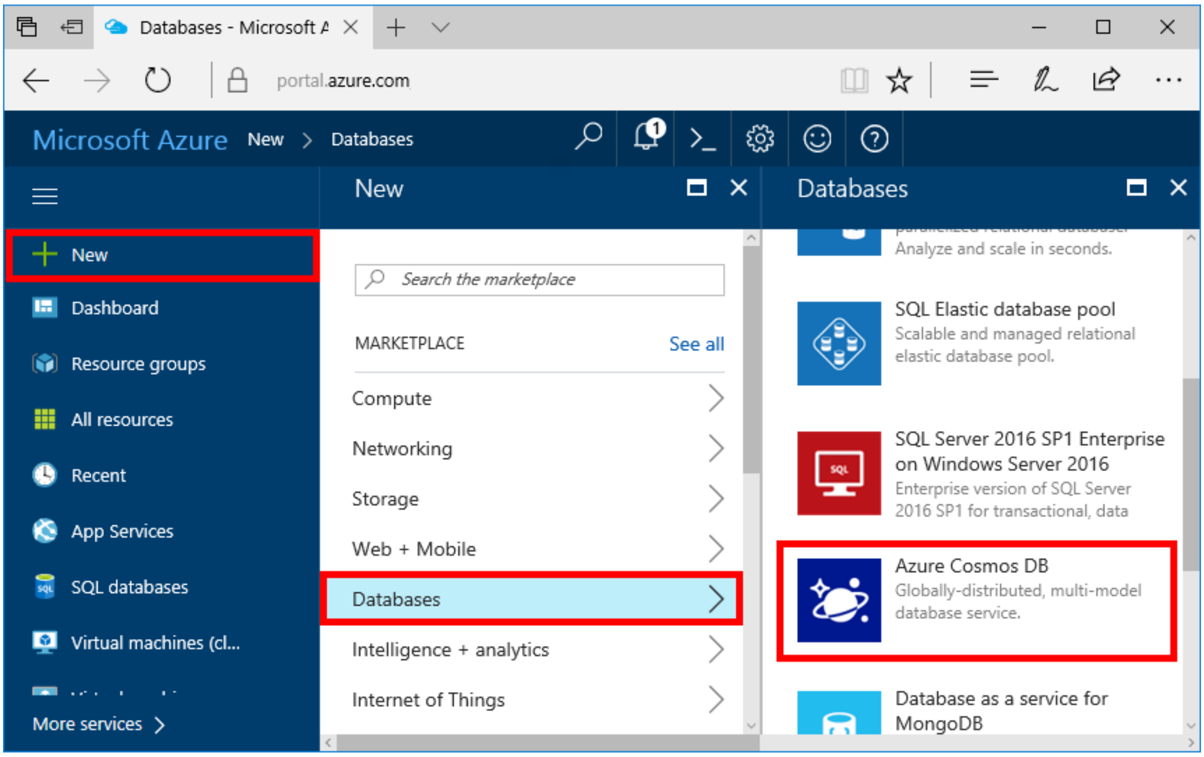
Prerequisites

If you don't already have Visual Studio 2017 installed, you can download and use the **free** Visual Studio 2017 Community Edition. Make sure that you enable **Azure development** during the Visual Studio setup.

If you don't have an Azure subscription, create a free account before you begin.

Create a database account

1. In a new window, sign in to the Azure portal.
2. In the left menu, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

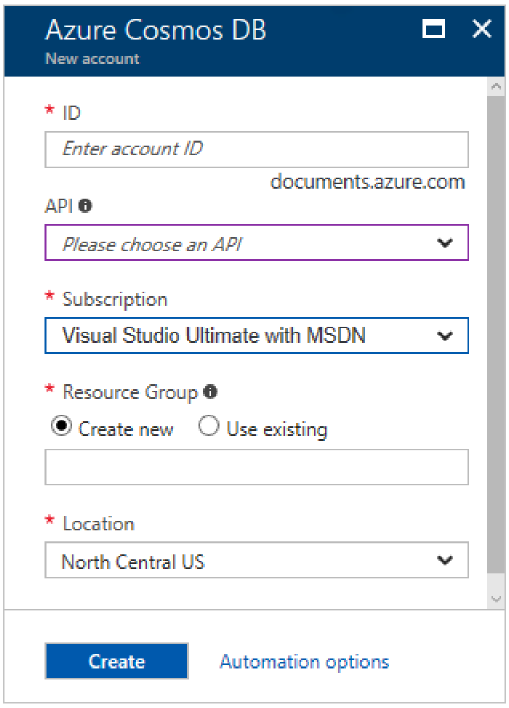


3. In the **New account** blade, specify the desired configuration for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick start we'll be programming against the MongoDB API so you'll choose **MongoDB** as you fill out the form. But if you have graph data for a social media app, document data from a catalog app, or key/value (table) data, realize that Azure Cosmos DB can provide a highly available, globally-distributed database service platform for all your mission-critical applications.

Fill out the New account blade using the information in the screenshot as a guide . You will choose unique values as you set up your account so your values will not match the screenshot exactly



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

* Subscription
Visual Studio Ultimate with MSDN

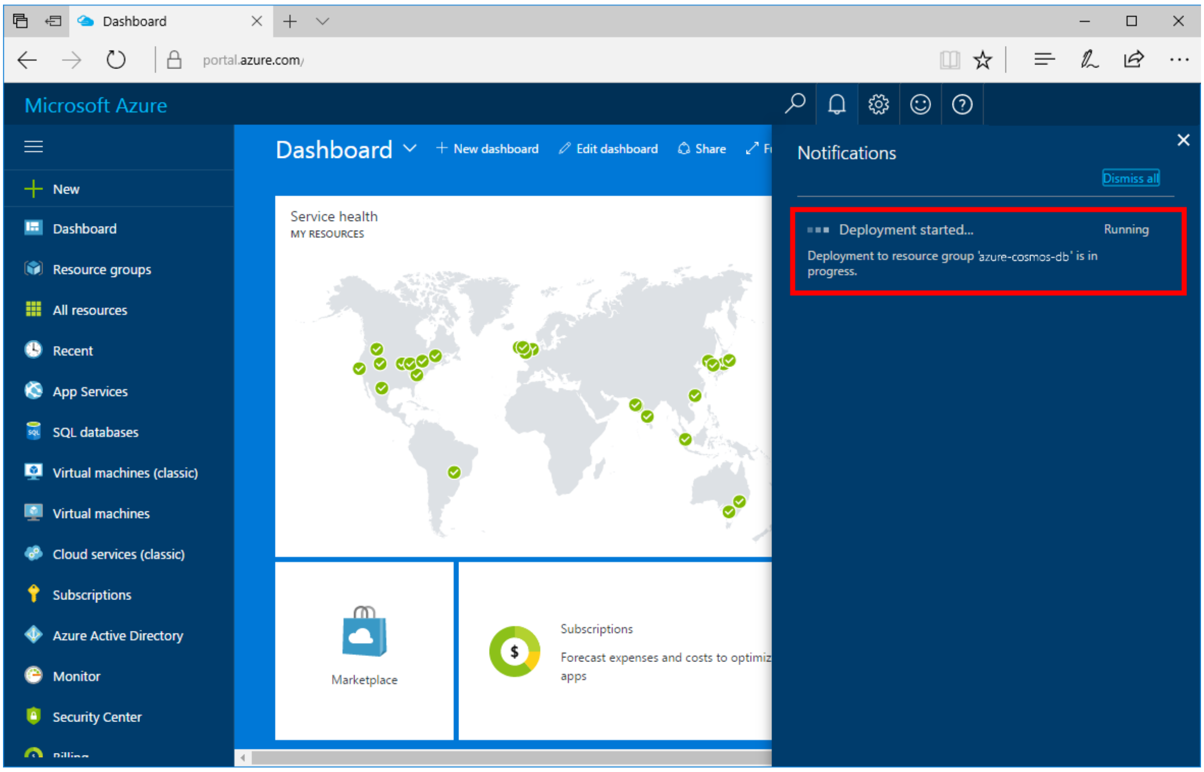
* Resource Group
● Create new ○ Use existing

* Location
North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name you choose to identify the Azure Cosmos DB account. *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID may contain only lowercase letters, numbers, and the '-' character, and must be between 3 and 50 characters.
API	MongoDB	We'll be programming against the MongoDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for the Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location closest to your users to give them the fastest access to the data.

4. Click **Create** to create the account.

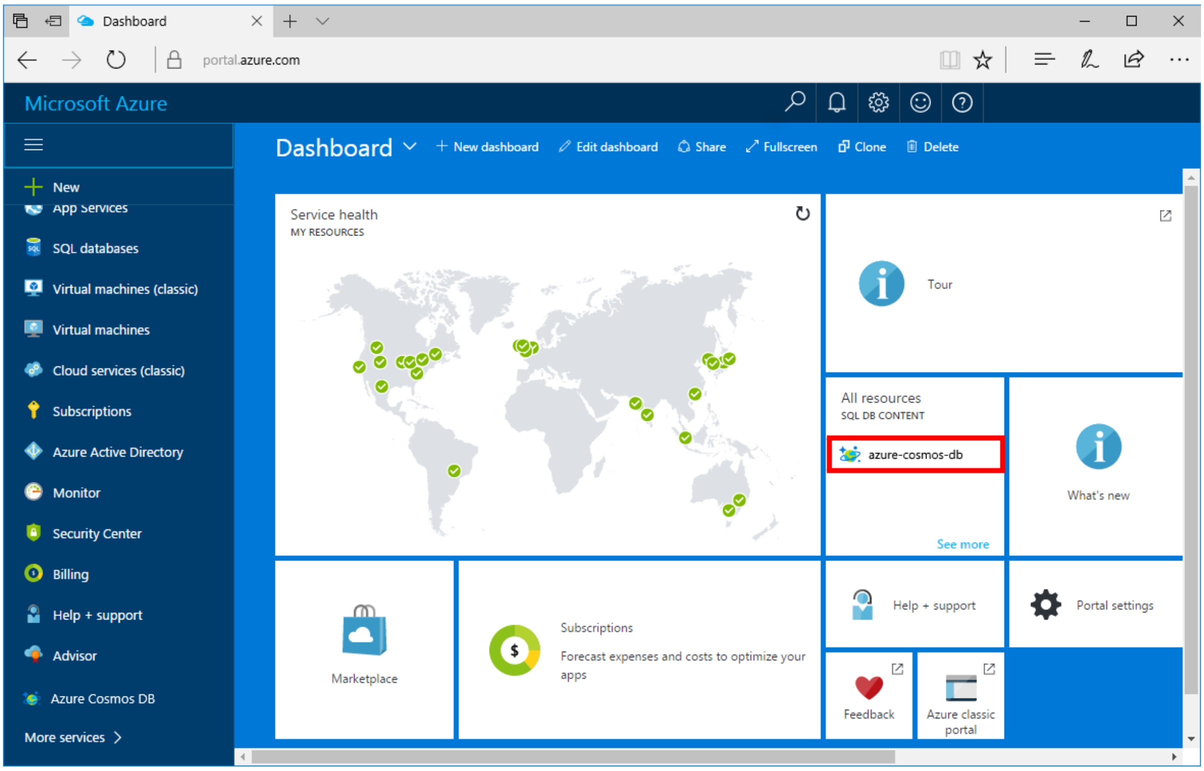
5. On the toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the All Resources tile.



Clone the sample application

Now let's clone a MongoDB API app from github, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a git terminal window, such as git bash, and `cd` to a working directory.

2. Run the following command to clone the sample repository.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-mongodb-dotnet-getting-started.git
```

3. Then open the solution file in Visual Studio.

Review the code

Let's make a quick review of what's happening in the app. Open the **Dal.cs** file under the **DAL** directory and you'll find that these lines of code create the Azure Cosmos DB resources.

- Initialize the Mongo Client.

```
  MongoClientSettings settings = new MongoClientSettings();
  settings.Server = new MongoServerAddress(host, 10255);
  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);
```

- Retrieve the database and the collection.

```
private string dbName = "Tasks";
private string collectionName = "TasksList";

var database = client.GetDatabase(dbName);
var todoTaskCollection = database.GetCollection<MyTask>(collectionName);
```

- Retrieve all documents.

```
collection.Find(new BsonDocument()).ToList();
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. In the Azure portal, in your Azure Cosmos DB account, in the left navigation click **Connection String**, and then click **Read-write Keys**. You'll use the copy buttons on the right side of the screen to copy the Username, Password, and Host into the Dal.cs file in the next step.

2. Open the **Dal.cs** file in the **DAL** directory.

3. Copy your **username** value from the portal (using the copy button) and make it the value of the **username** in your **Dal.cs** file.

4. Then copy your **host** value from the portal and make it the value of the **host** in your **Dal.cs** file.

5. Finally copy your **password** value from the portal and make it the value of the **password** in your **Dal.cs** file.

You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

Run the web app

1. In Visual Studio, right-click on the project in **Solution Explorer** and then click **Manage NuGet Packages**.

2. In the NuGet **Browse** box, type *MongoDB.Driver*.

3. From the results, install the **MongoDB.Driver** library. This installs the MongoDB.Driver package as well as all dependencies.

4. Click CTRL + F5 to run the application. Your app displays in your browser.

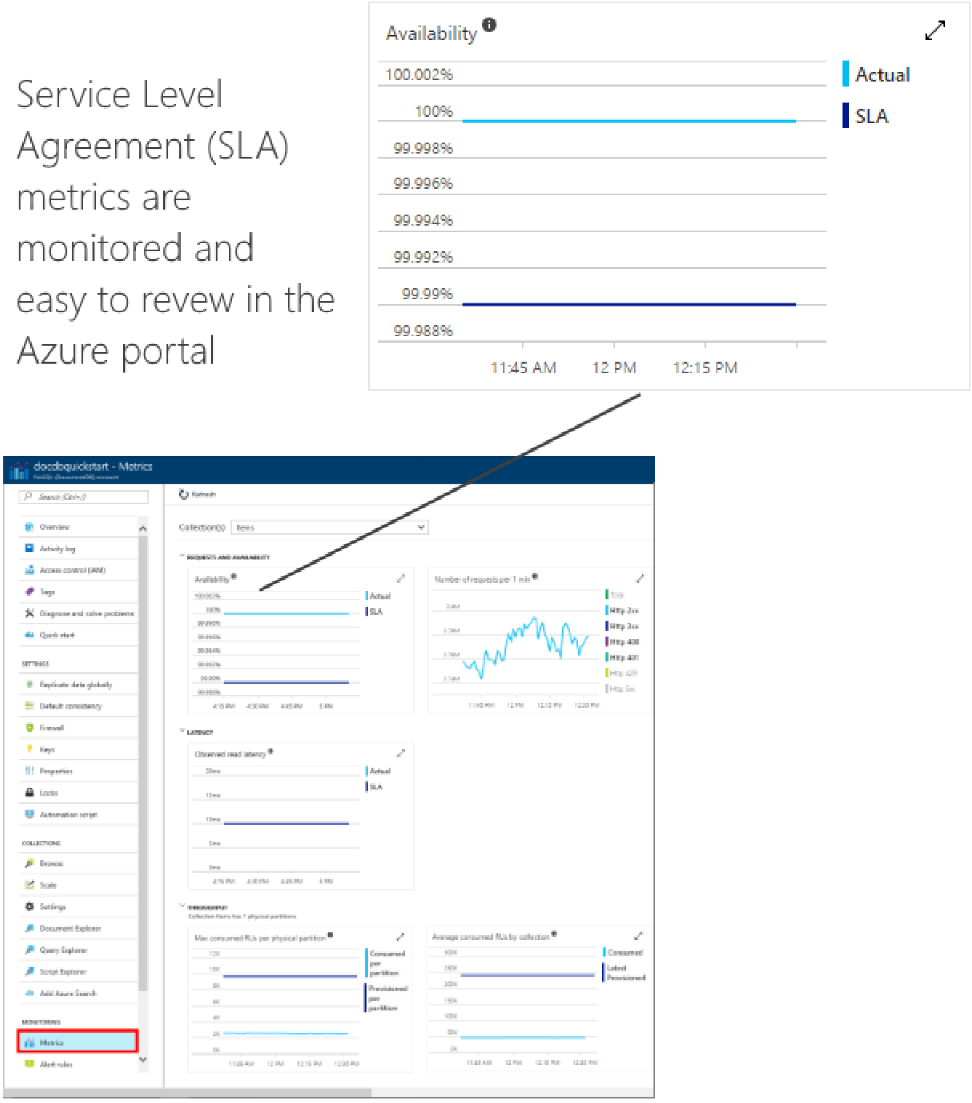
5. Click **Create** in the browser and create a few new tasks in your task list app.

Review SLAs in the Azure portal

Now that your app is up and running, you'll want to ensure business continuity and watch user access to ensure high availability. You can use the Azure portal to review the availability, latency, throughput, and consistency of your collection.

Each graph that's associated with the Azure Cosmos DB Service Level Agreements (SLAs) provides a line that shows the quota required to meet the SLA and your actual usage, giving you a clear view into your database performance. Additional metrics, such as storage usage and number of requests per minute, are also included in the portal.

- In the Azure portal, in the left pane, under **Monitoring**, click **Metrics**.



Service Level Agreement (SLA) metrics are monitored and easy to revew in the the Azure portal

Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart in the Azure portal with the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account and run a web app using the API for MongoDB. You can now import additional data to your Cosmos DB account.

Import data into Azure Cosmos DB for the MongoDB API

Azure Cosmos DB: Build a MongoDB API console app with Java and the Azure portal

6/1/2017 • 5 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quick start demonstrates how to create an Azure Cosmos DB account, document database, and collection using the Azure portal. You'll then build and deploy a console app built on the MongoDB Java driver.

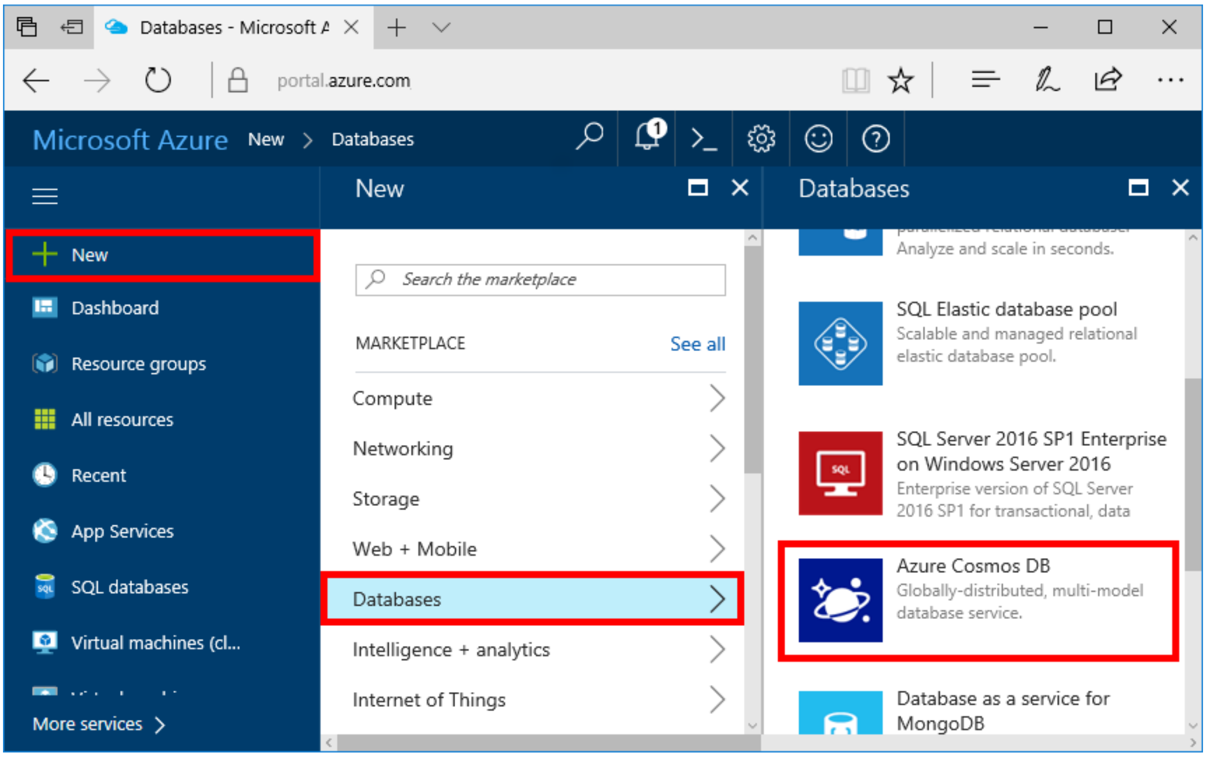
Prerequisites

- Before you can run this sample, you must have the following prerequisites:
  - JDK 1.7+ (Run `apt-get install default-jdk` if you don't have JDK)
  - Maven (Run `apt-get install maven` if you don't have Maven)

If you don't have an Azure subscription, create a free account before you begin.

Create a database account

1. In a new window, sign in to the Azure portal.
2. In the left menu, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

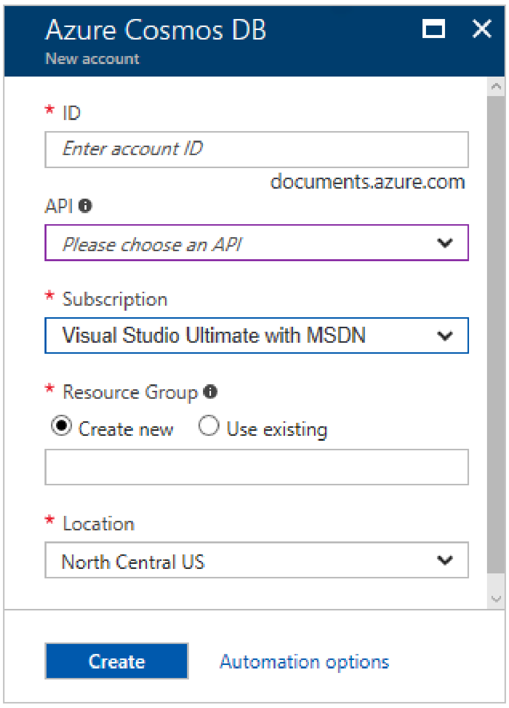


3. In the **New account** blade, specify the desired configuration for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick start we'll be programming against the MongoDB API so you'll choose **MongoDB** as you fill out the form. But if you have graph data for a social media app, document data from a catalog app, or key/value (table) data, realize that Azure Cosmos DB can provide a highly available, globally-distributed database service platform for all your mission-critical applications.

Fill out the New account blade using the information in the screenshot as a guide . You will choose unique values as you set up your account so your values will not match the screenshot exactly



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

* Subscription
Visual Studio Ultimate with MSDN

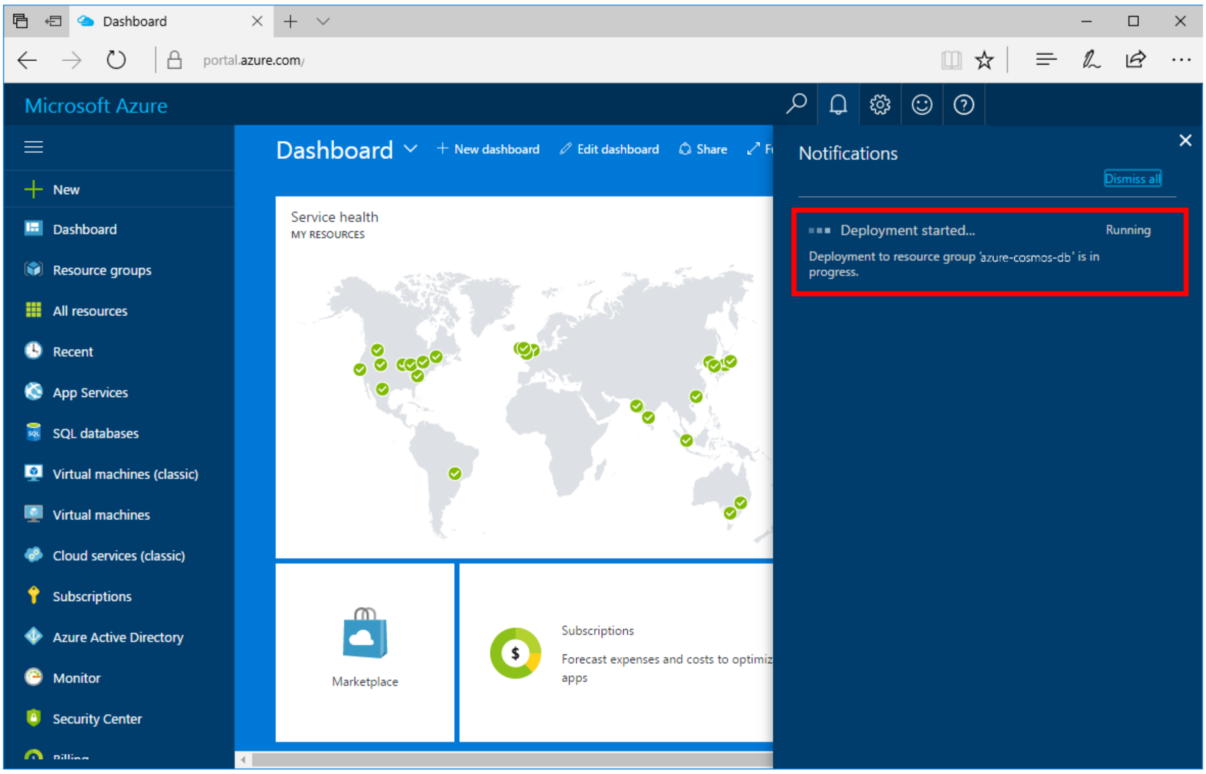
* Resource Group
● Create new ○ Use existing

* Location
North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name you choose to identify the Azure Cosmos DB account. *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID may contain only lowercase letters, numbers, and the '-' character, and must be between 3 and 50 characters.
API	MongoDB	We'll be programming against the MongoDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for the Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location closest to your users to give them the fastest access to the data.

4. Click **Create** to create the account.

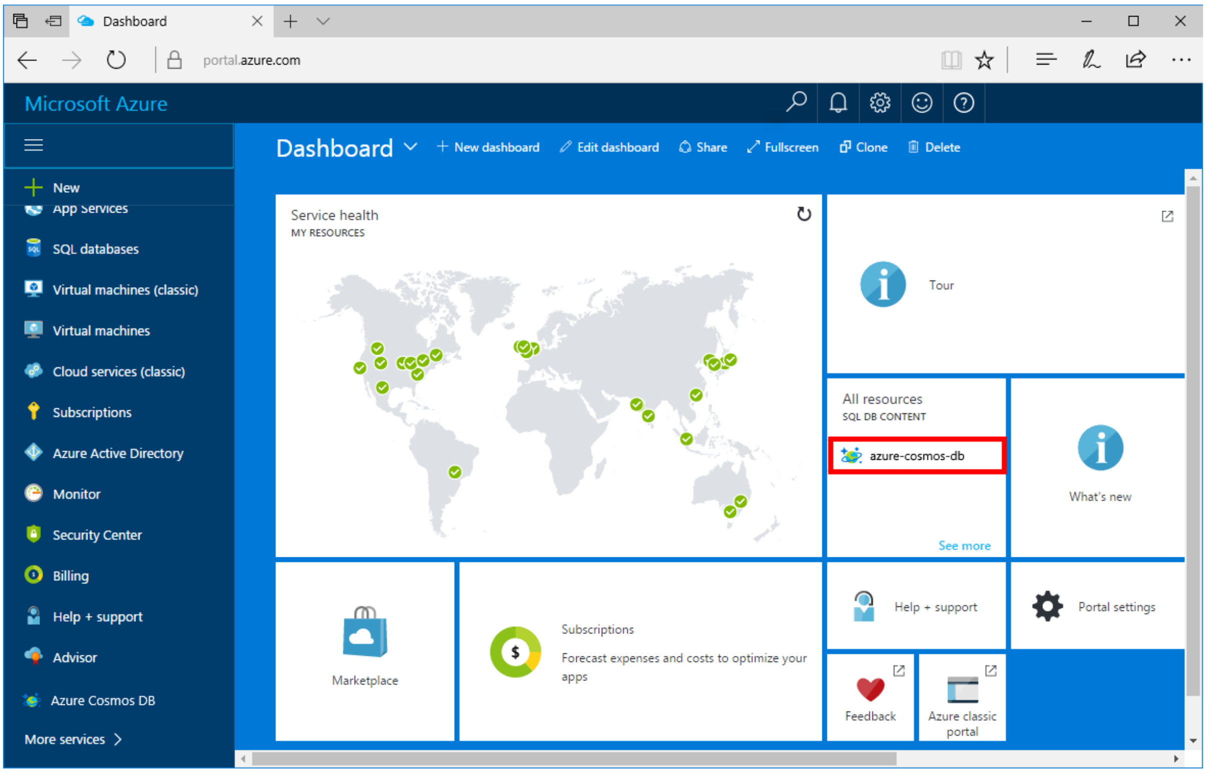
5. On the toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the All Resources tile.



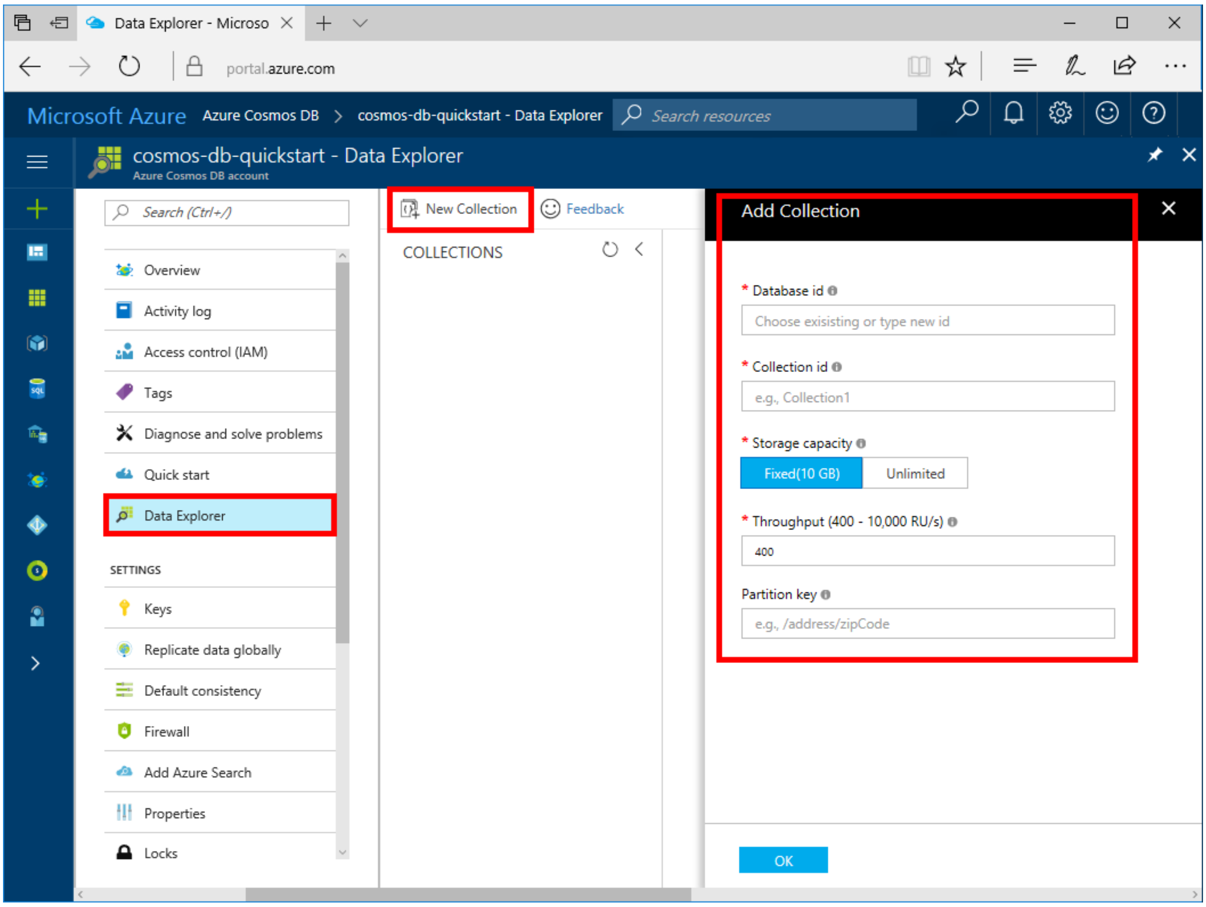
Add a collection

Name your new database, **db**, and your new collection, **coll**.

You can now use Data Explorer to create a collection and add data to your database.

1. In the Azure portal, in the left pane, click **Data Explorer**.

2. On the **Data Explorer** blade, click **New Collection**, and then provide the following information:



Add Collection

* Database id
Choose exisisting or type new id

* Collection id
e.g., Collection1

* Storage capacity
Fixed(10 GB) | Unlimited

* Throughput (400 - 10,000 RU/s)
400

Partition key
e.g., /address/zipCode

SETTING	SUGGESTED VALUE	DESCRIPTION
Database id	Items	The ID for your new database. Database names must contain from 1 through 255 characters, and they cannot contain /, \, #, ?, or a trailing space.
Collection id	ToDoList	The ID for your new collection. Collection names have the same character requirements as database IDs.
Storage capacity	Fixed (10 GB)	Use the default value. This is the storage capacity of the database.
Throughput	400 RU	Use the default value. If you want to reduce latency, you can scale up the throughput later.
Partition key	/userid	A partition key that distributes data evenly to each partition. Selecting the correct partition key is important in creating a performant collection. To learn more, see Designing for partitioning.

3. After you've completed the form, click **OK**.

Clone the sample application

Now let's clone a MongoDB API app from github, set the connection string, and run it. You'll see how easy it is to

work with data programmatically.

1. Open a git terminal window, such as git bash, and `cd` to a working directory.

2. Run the following command to clone the sample repository.

   ```
   git clone https://github.com/Azure-Samples/azure-cosmos-db-mongodb-java-getting-started.git
   ```

3. Then open the solution file in Visual Studio.

Review the code

Let's make a quick review of what's happening in the app. Open the `Program.cs` file and you'll find that these lines of code create the Azure Cosmos DB resources.

- The DocumentClient is initialized.

  ```
  MongoClientURI uri = new MongoClientURI("FILLME");`

  MongoClient mongoClient = new MongoClient(uri);
  ```

- A new database and collection are created.

  ```
  MongoDatabase database = mongoClient.getDatabase("db");

  MongoCollection<Document> collection = database.getCollection("coll");
  ```

- Some documents are inserted using `MongoCollection.insertOne`

  ```
  Document document = new Document("fruit", "apple")
  collection.insertOne(document);
  ```

- Some queries are performed using `MongoCollection.find`

  ```
  Document queryResult = collection.find(Filters.eq("fruit", "apple")).first();
  System.out.println(queryResult.toJson());
  ```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. From the Account, select **Quick Start**, select Java, then copy the connection string to your clipboard

2. Open the `Program.java` file, replace the argument to the MongoClientURI constructor with the connection string. You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

Run the console app

1. Run `mvn package` in a terminal to install required npm modules

2. Run `mvn exec:java -D exec.mainClass=GetStarted.Program` in a terminal to start your Java application.

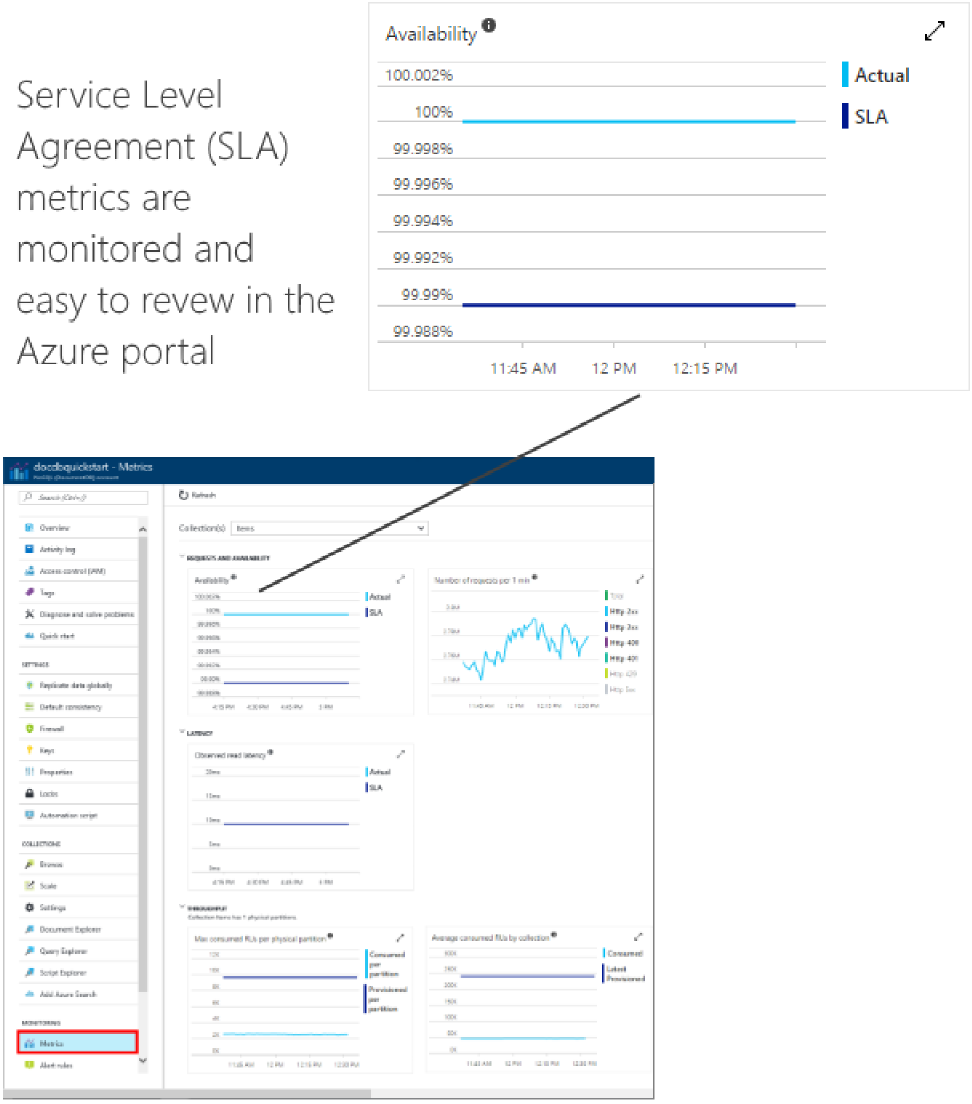
You can now use Robomongo / Studio 3T to query, modify, and work with this new data.

Review SLAs in the Azure portal

Now that your app is up and running, you'll want to ensure business continuity and watch user access to ensure high availability. You can use the Azure portal to review the availability, latency, throughput, and consistency of your collection.

Each graph that's associated with the Azure Cosmos DB Service Level Agreements (SLAs) provides a line that shows the quota required to meet the SLA and your actual usage, giving you a clear view into your database performance. Additional metrics, such as storage usage and number of requests per minute, are also included in the portal.

- In the Azure portal, in the left pane, under **Monitoring**, click **Metrics**.



Service Level Agreement (SLA) metrics are monitored and easy to revew in the Azure portal

Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart in the Azure portal with the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a collection using the Data Explorer, and run a console app. You can now import additional data to your Cosmos DB account.

Import MongoDB data into Azure Cosmos DB

Azure Cosmos DB: Build a .NET application using the Graph API

6/6/2017 • 7 min to read • Edit Online

Azure Cosmos DB is Microsoft’s globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quick start demonstrates how to create an Azure Cosmos DB account, database, and graph (container) using the Azure portal. You then build and run a console app built on the Graph API (preview).

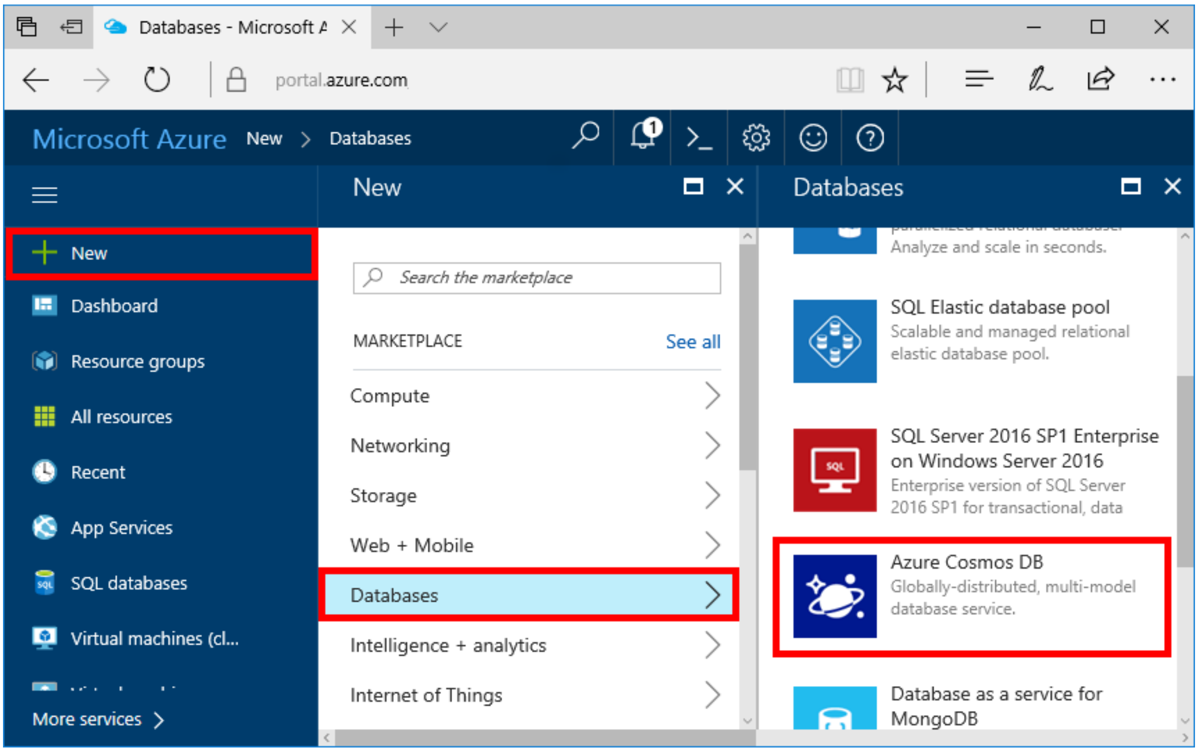
Prerequisites

If you don’t already have Visual Studio 2017 installed, you can download and use the **free** Visual Studio 2017 Community Edition. Make sure that you enable **Azure development** during the Visual Studio setup.

If you don't have an Azure subscription, create a free account before you begin.

Create a database account

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

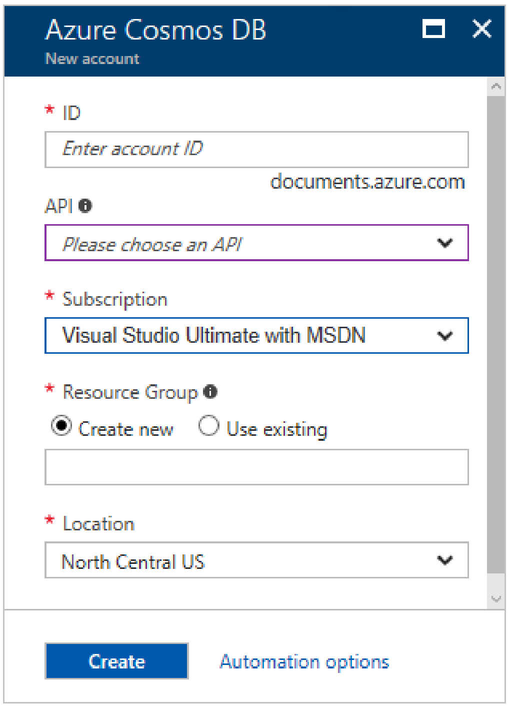


3. In the **New account** blade, specify the desired configuration for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article, we program against the Graph API, so choose **Gremlin (graph)** as you fill out the form. If you have document data from a catalog app, key/value (table) data, or data that's migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

On the **New account** blade, complete the fields with the information in the following screenshot as a guide only. Because your own values will not match those in the screenshot, be sure to choose unique values as you set up your account.



Azure Cosmos DB

New account

* ID

Enter account ID

documents.azure.com

API

Please choose an API

* Subscription

Visual Studio Ultimate with MSDN

* Resource Group

● Create new ○ Use existing

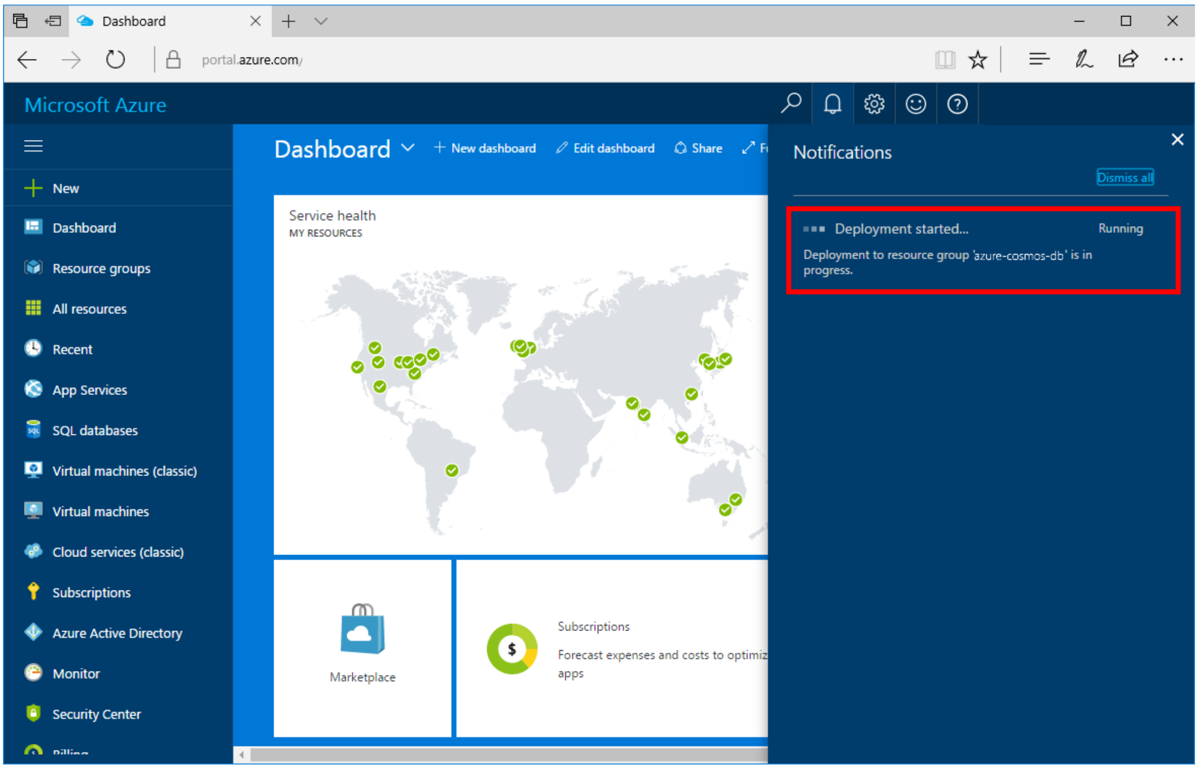
* Location

North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that you choose to identify the Azure Cosmos DB account. Because *documents.azure.com* is appended to the ID that you provide to create your URI, use a unique but identifiable ID. The ID must contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 to 50 characters.
API	Gremlin (graph)	We program against the Graph API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for the Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location closest to your users to give them the fastest access to the data.

4. Click **Create** to create the account.

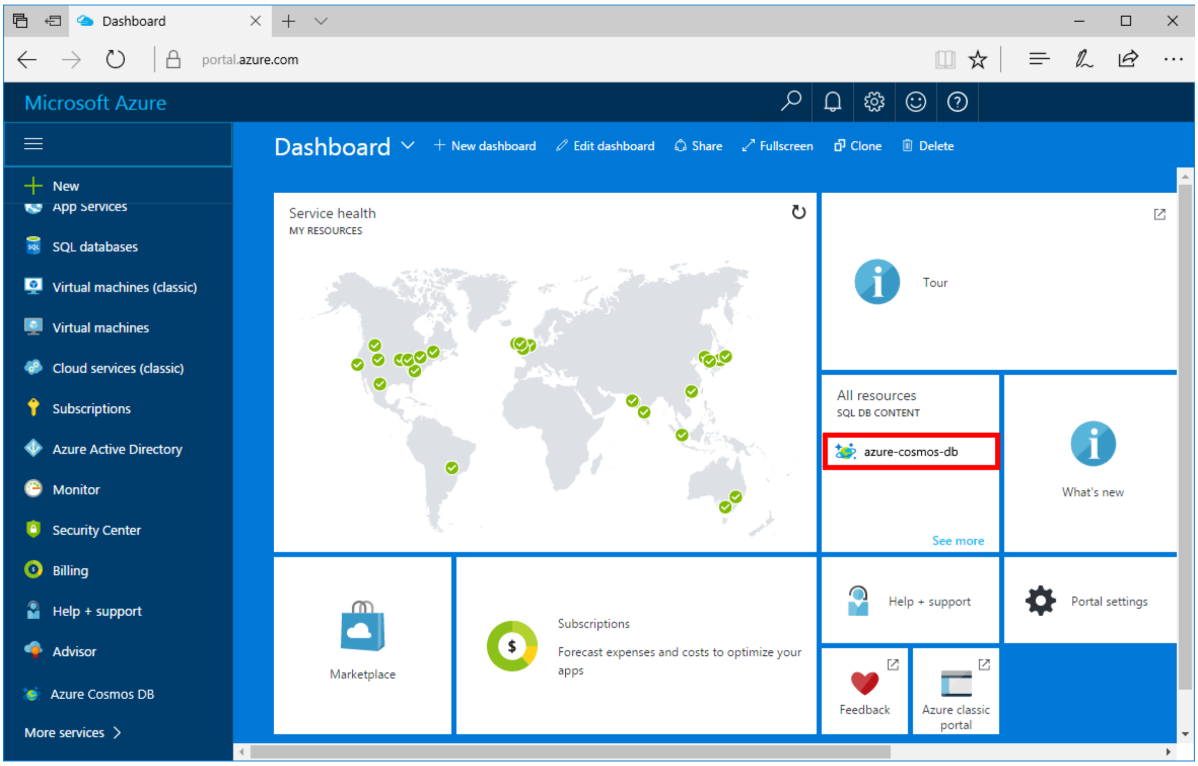
5. On the toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

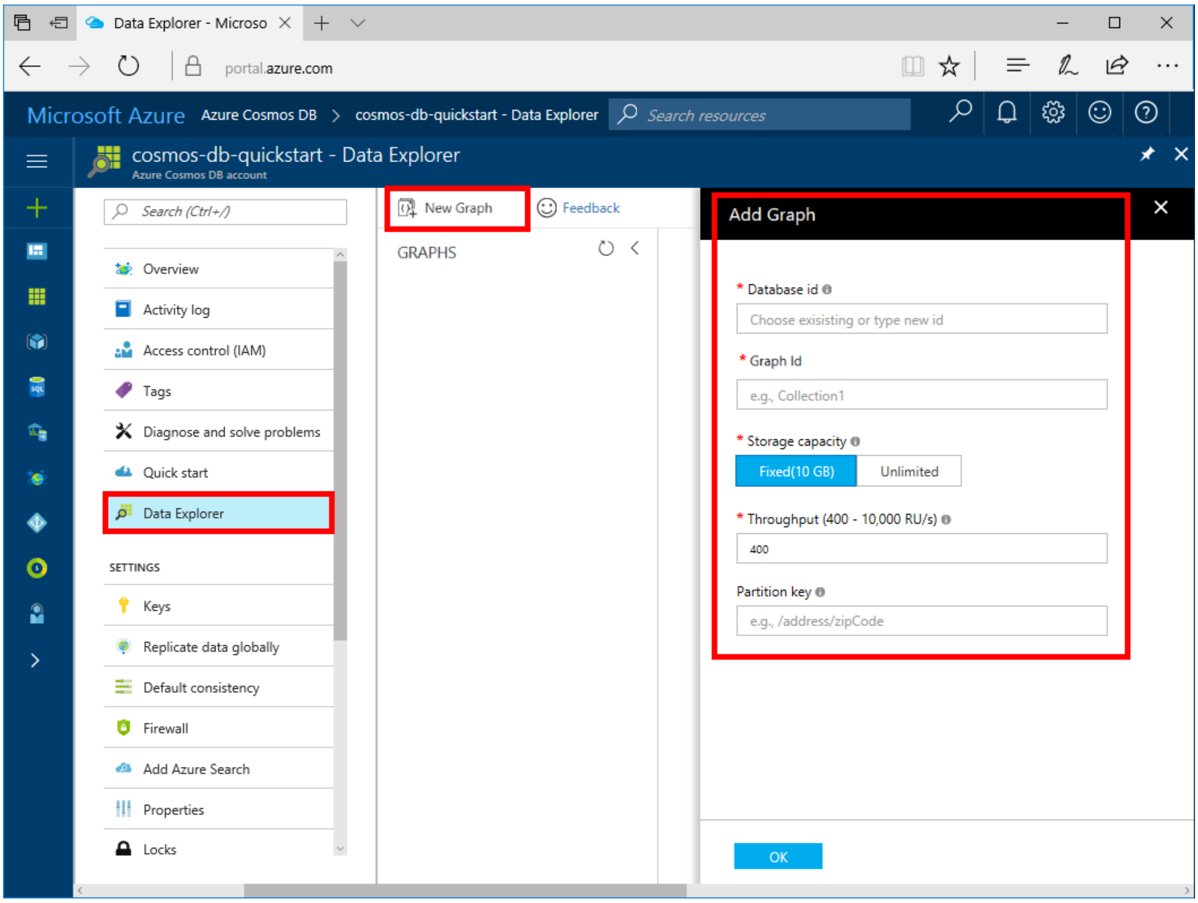
6. When the deployment is complete, open the new account from the **All Resources** tile.



Add a graph

You can now use Data Explorer to create a graph container and add data to your database.

1. In the Azure portal, in the navigation menu, click **Data Explorer**.
2. In the Data Explorer blade, click **New Graph**, then fill in the page using the following information.



Add Graph

* Database id

Choose exisisting or type new id

* Graph Id

e.g., Collection1

* Storage capacity

Fixed(10 GB) | Unlimited

* Throughput (400 - 10,000 RU/s)

400

Partition key

e.g., /address/zipCode

OK

SETTING	SUGGESTED VALUE	DESCRIPTION
Database id	sample-database	The ID for your new database. Database names must be between 1 and 255 characters, and cannot contain `/ \ # ?` or a trailing space.
Graph id	sample-graph	The ID for your new graph. Graph names have the same character requirements as database ids.
Storage Capacity	10 GB	Leave the default value. This is the storage capacity of the database.
Throughput	400 RUs	Leave the default value. You can scale up the throughput later if you want to reduce latency.
Partition key	/userid	A partition key that will distribute data evenly to each partition. Selecting the correct partition key is important in creating a performant graph, read more about it in Designing for partitioning.

3. Once the form is filled out, click **OK**.

Clone the sample application

Now let's clone a Graph API app from github, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a git terminal window, such as git bash, and `cd` to a working directory.
2. Run the following command to clone the sample repository.

   ```
   git clone https://github.com/Azure-Samples/azure-cosmos-db-graph-dotnet-getting-started.git
   ```

3. Then open the solution file in Visual Studio.

Review the code

Let's make a quick review of what's happening in the app. Open the Program.cs file and you'll find that these lines of code create the Azure Cosmos DB resources.

- The DocumentClient is initialized. In the preview, we added a graph extension API on the Azure Cosmos DB client. We are working on a standalone graph client decoupled from the Azure Cosmos DB client and resources.

  ```
  using (DocumentClient client = new DocumentClient(
     new Uri(endpoint),
     authKey,
     new ConnectionPolicy { ConnectionMode = ConnectionMode.Direct, ConnectionProtocol = Protocol.Tcp }))
  ```

- A new database is created.

  ```
  Database database = await client.CreateDatabaseIfNotExistsAsync(new Database { Id = "graphdb" });
  ```

- A new graph is created.

  ```
  DocumentCollection graph = await client.CreateDocumentCollectionIfNotExistsAsync(
     UriFactory.CreateDatabaseUri("graphdb"),
     new DocumentCollection { Id = "graph" },
     new RequestOptions { OfferThroughput = 1000 });
  ```

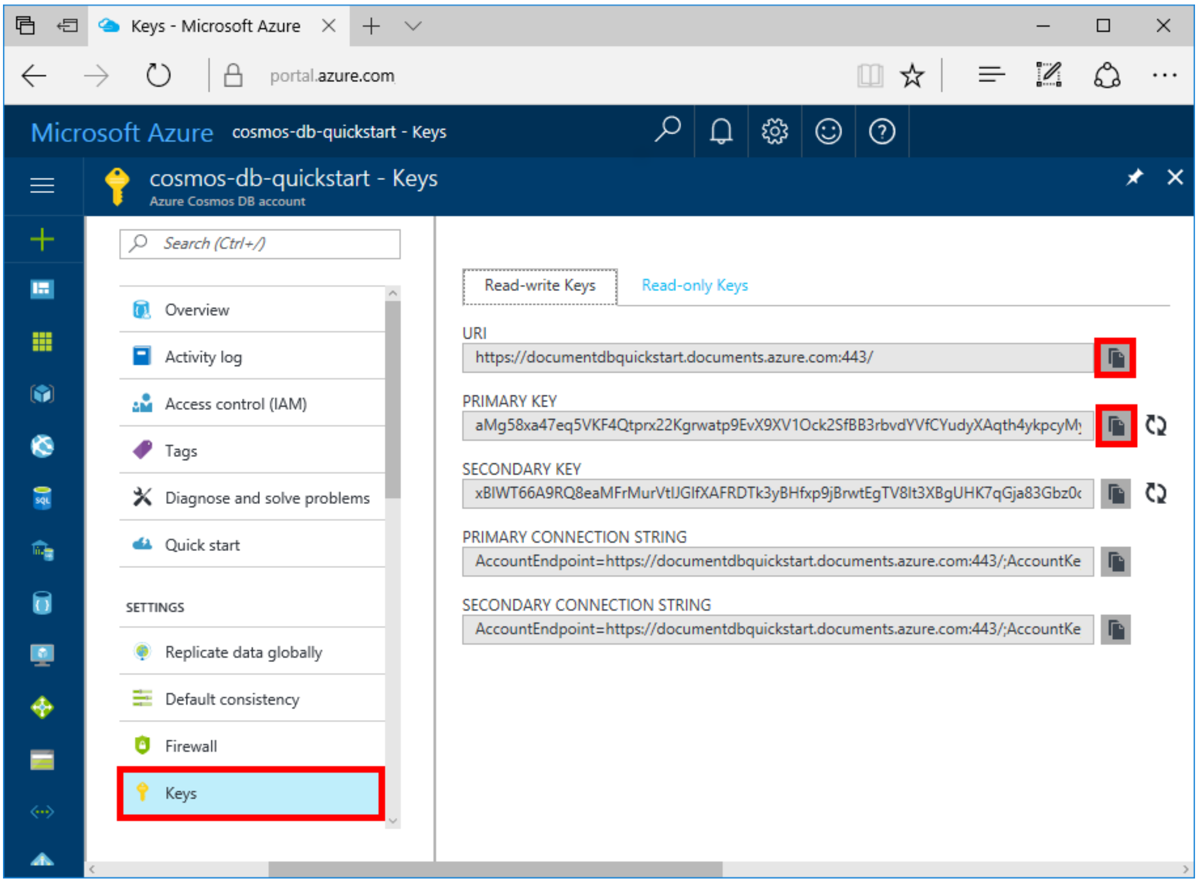
- A series of Gremlin steps are executed using the `CreateGremlinQuery` method.

  ```
  // The CreateGremlinQuery method extensions allow you to execute Gremlin queries and iterate
  // results asychronously
  IDocumentQuery<dynamic> query = client.CreateGremlinQuery<dynamic>(graph, "g.V().count()");
  while (query.HasMoreResults)
  {
     foreach (dynamic result in await query.ExecuteNextAsync())
     {
        Console.WriteLine($"\t {JsonConvert.SerializeObject(result)}");
     }
  }
  ```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. In the Azure portal, in your Azure Cosmos DB account, in the left navigation click **Keys**, and then click **Read-write Keys**. You'll use the copy buttons on the right side of the screen to copy the URI and Primary Key into the `App.config` file in the next step.



URI
https://documentdbquickstart.documents.azure.com:443/

PRIMARY KEY
aMg58xa47eq5VKF4Qtprx22Kgrwatp9EvX9XV1Ock2SfBB3rbvdYVfCYudyXAqth4ykpcyMy

SECONDARY KEY
xBIWT66A9RQ8eaMFrMurVtIJGIfXAFRDTk3yBHfxp9jBrwtEgTV8It3XBgUHK7qGja83Gbz0c

PRIMARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

SECONDARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

2. In Visual Studio 2017, open the `App.config` file.

3. Copy your URI value from the portal (using the copy button) and make it the value of the endpoint key in `App.config`.

   ```
   <add key="Endpoint" value="FILLME.documents.azure.com:443" />
   ```

4. Then copy your PRIMARY KEY value from the portal and make it the value of the authKey in `App.config`.

   ```
   <add key="AuthKey" value="FILLME" />
   ```

You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

Run the console app

1. In Visual Studio, right-click on the **GraphGetStarted** project in **Solution Explorer** and then click **Manage NuGet Packages**.

2. In the NuGet **Browse** box, type *Microsoft.Azure.Graphs* and check the **Includes prerelease** box.

3. From the results, install the **Microsoft.Azure.Graphs** library. This installs the Azure Cosmos DB graph extension library package and all dependencies.

4. Click CTRL + F5 to run the application.

   The console window displays the vertexes and edges being added to the graph. When the script completes, press ENTER twice to close the console window.

Browse using the Data Explorer

You can now go back to Data Explorer in the Azure portal and browse and query your new graph data.

- In Data Explorer, the new database appears in the Collections pane. Expand **graphdb**, **graphcoll**, and then click **Graph**.

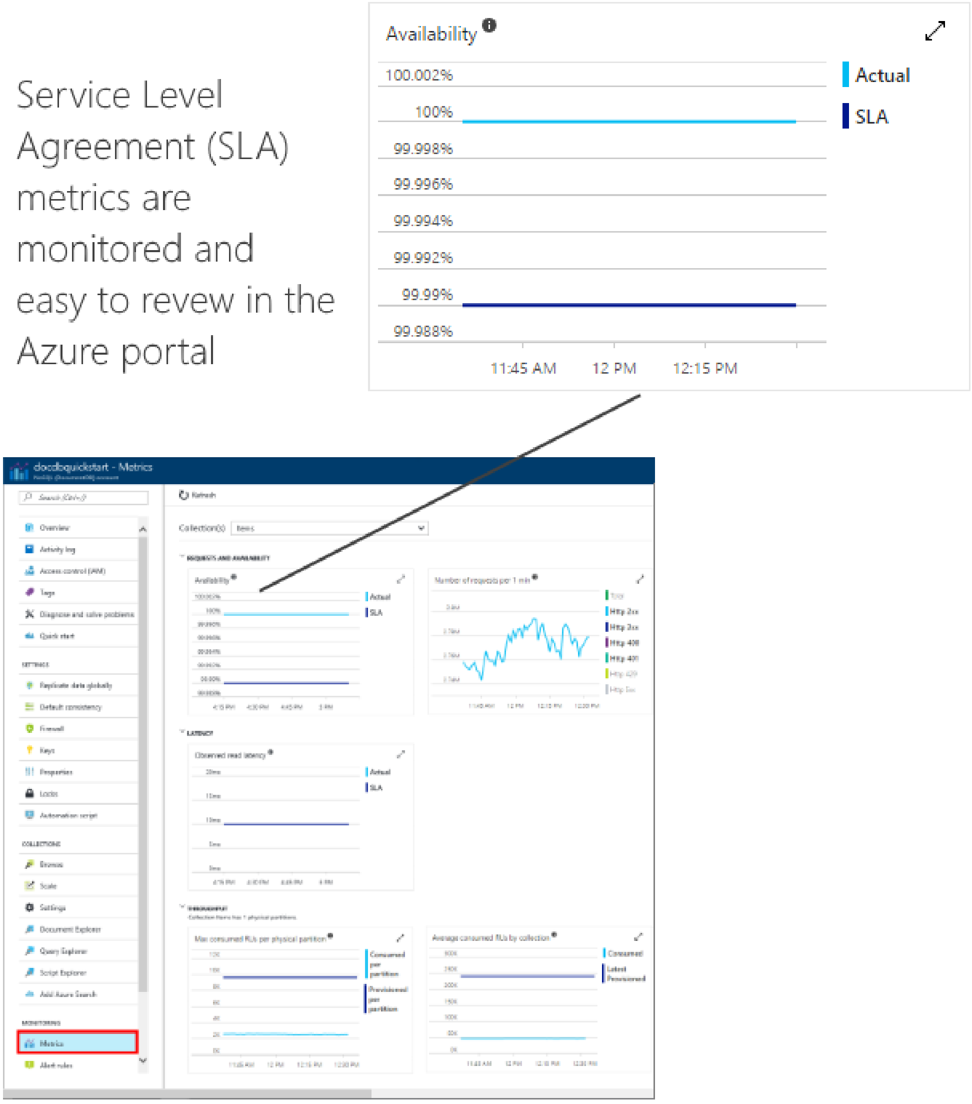
The data generated by the sample app is displayed in the Graphs pane.

Review SLAs in the Azure portal

Now that your app is up and running, you'll want to ensure business continuity and watch user access to ensure high availability. You can use the Azure portal to review the availability, latency, throughput, and consistency of your collection.

Each graph that's associated with the Azure Cosmos DB Service Level Agreements (SLAs) provides a line that shows the quota required to meet the SLA and your actual usage, giving you a clear view into your database performance. Additional metrics, such as storage usage and number of requests per minute, are also included in the portal.

- In the Azure portal, in the left pane, under **Monitoring**, click **Metrics**.



Service Level Agreement (SLA) metrics are monitored and easy to revew in the Azure portal

Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart in the Azure portal with the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a graph using the Data Explorer, and run an app. You can now build more complex queries and implement powerful graph traversal logic using Gremlin.

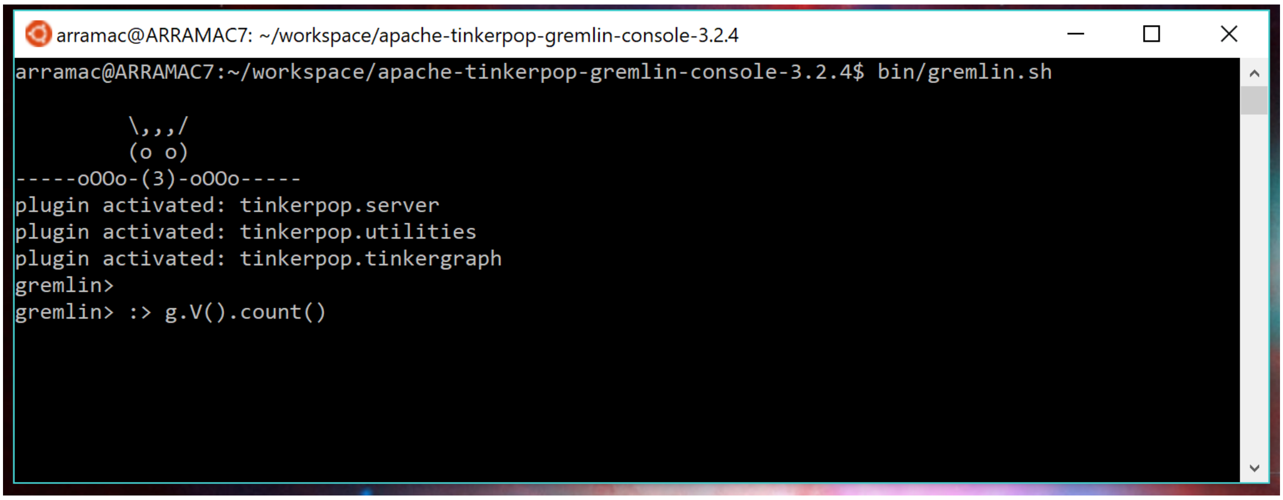
Query using Gremlin

Azure Cosmos DB: Create, query, and traverse a graph in the Gremlin console

6/12/2017 • 7 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quick start demonstrates how to create an Azure Cosmos DB account, database, and graph (container) using the Azure portal and then use the Gremlin Console from Apache TinkerPop to work with Graph API (preview) data. In this tutorial, you'll create and query vertices and edges, updating a vertex property, query vertices, traverse the graph, and drop a vertex.



arramac@ARRAMAC7: ~/workspace/apache-tinkerpop-gremlin-console-3.2.4

```
arramac@ARRAMAC7:~/workspace/apache-tinkerpop-gremlin-console-3.2.4$ bin/gremlin.sh

         \,,,/
         (o o)
-----oOOo-(3)-oOOo-----
plugin activated: tinkerpop.server
plugin activated: tinkerpop.utilities
plugin activated: tinkerpop.tinkergraph
gremlin>
gremlin> :> g.V().count()
```

The Gremlin console is Groovy/Java based and runs on Linux, Mac, and Windows. You can download it from the Apache TinkerPop site.

Prerequisites

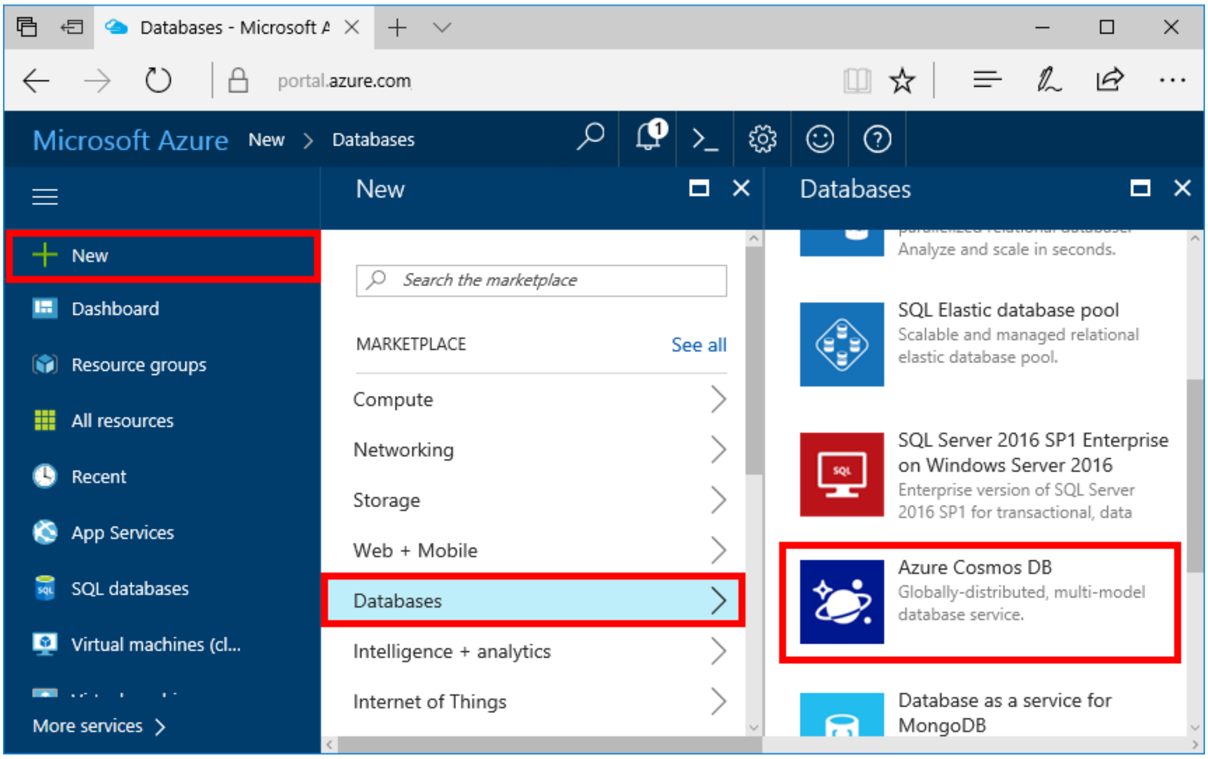
You need to have an Azure subscription to create an Azure Cosmos DB account for this quickstart.

If you don't have an Azure subscription, create a free account before you begin.

You also need to install the Gremlin Console. Use version 3.2.4 or above.

Create a database account

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

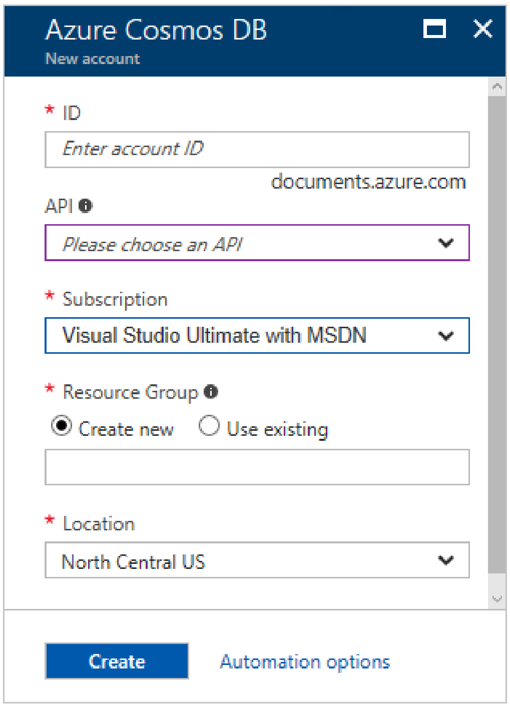


3. In the **New account** blade, specify the desired configuration for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article, we program against the Graph API, so choose **Gremlin (graph)** as you fill out the form. If you have document data from a catalog app, key/value (table) data, or data that's migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

   On the **New account** blade, complete the fields with the information in the following screenshot as a guide only. Because your own values will not match those in the screenshot, be sure to choose unique values as you set up your account.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

* Subscription
Visual Studio Ultimate with MSDN

* Resource Group
● Create new ○ Use existing

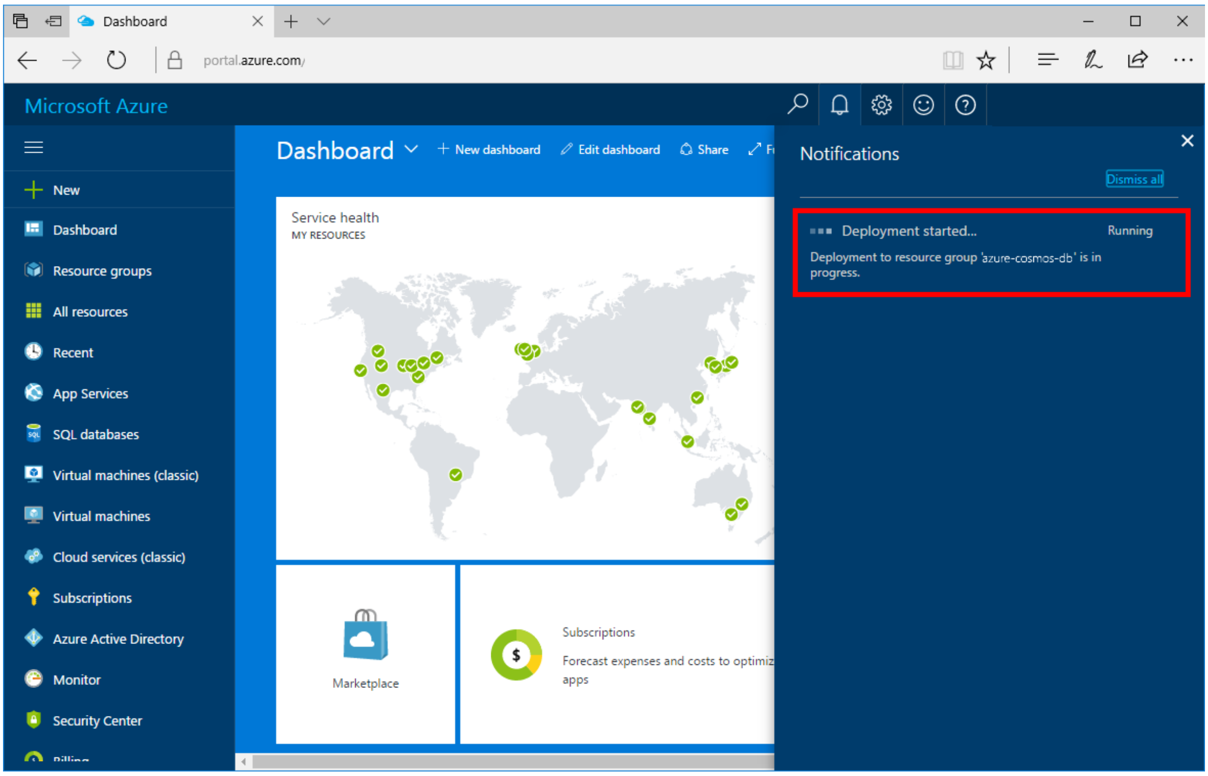
* Location
North Central US

Create   Automation options

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that you choose to identify the Azure Cosmos DB account. Because *documents.azure.com* is appended to the ID that you provide to create your URI, use a unique but identifiable ID. The ID must contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 to 50 characters.
API	Gremlin (graph)	We program against the Graph API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for the Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location closest to your users to give them the fastest access to the data.

4. Click **Create** to create the account.

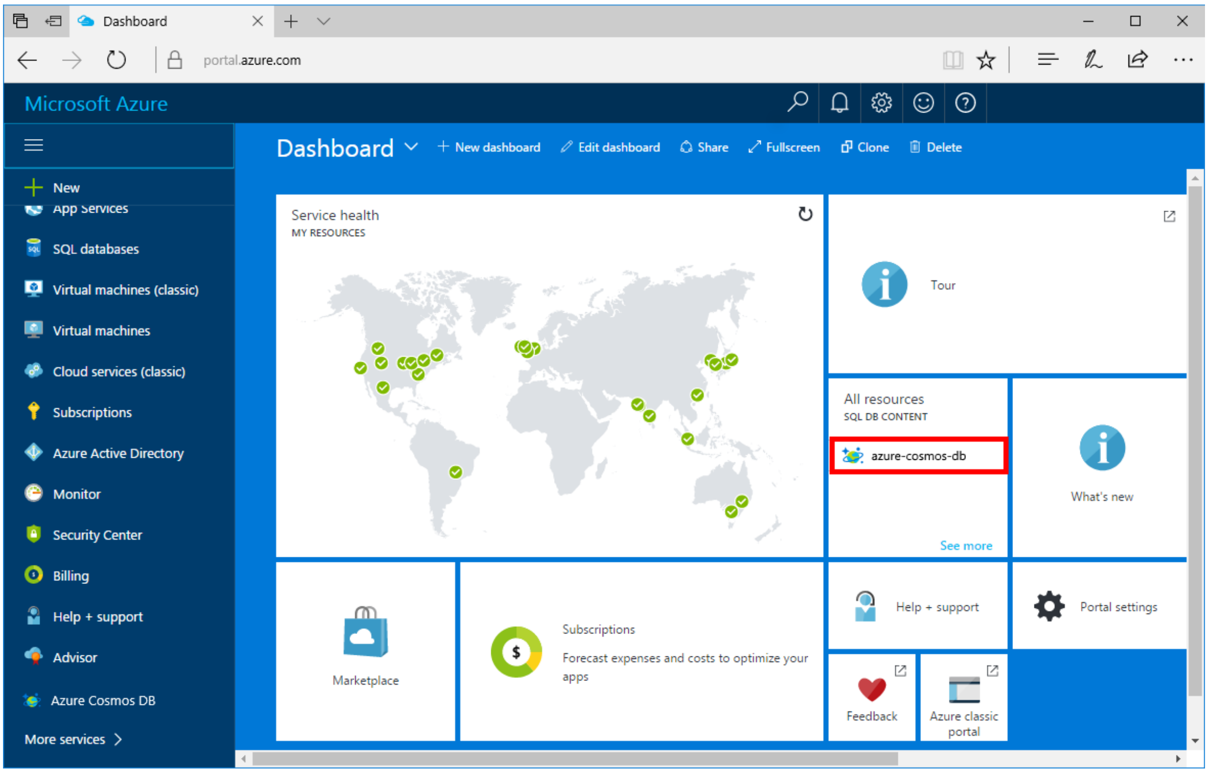
5. On the toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

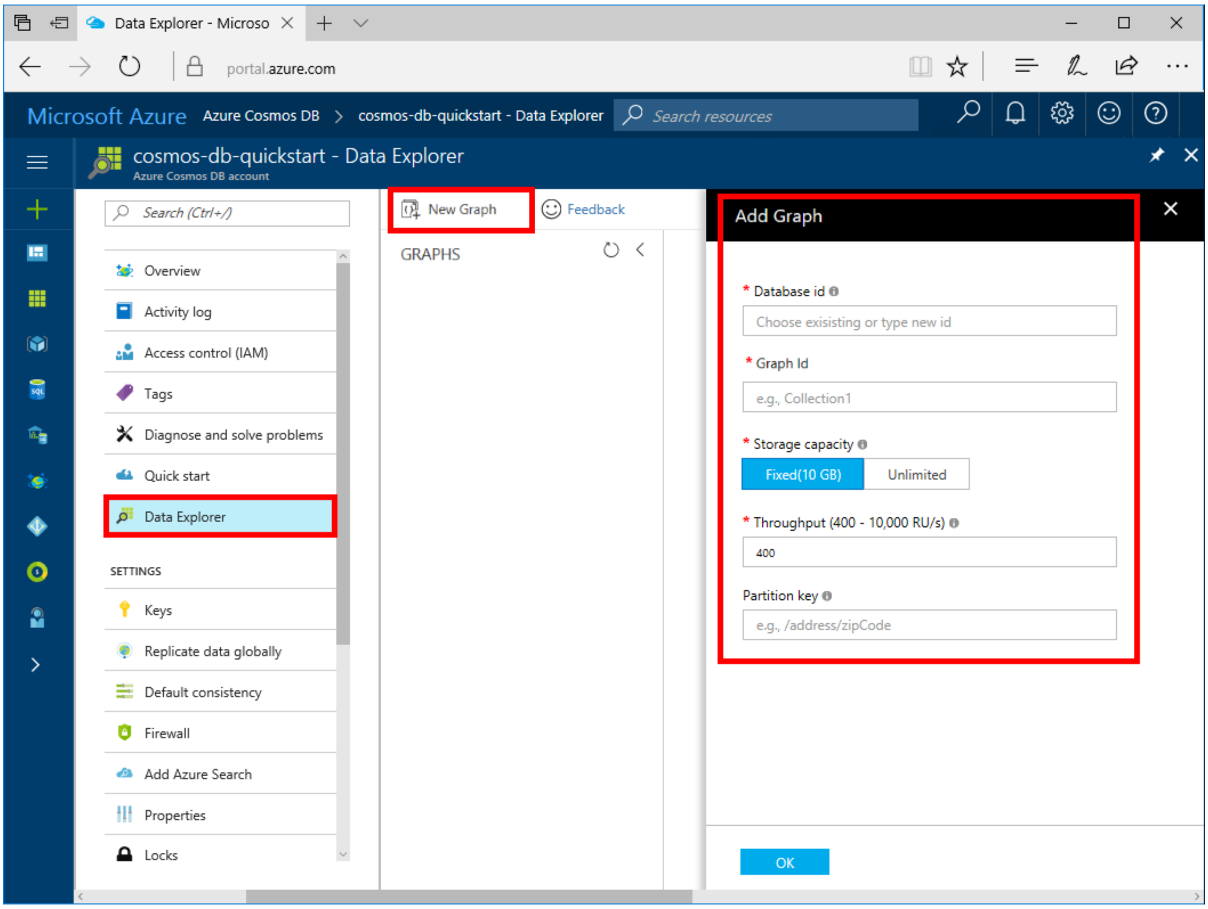
6. When the deployment is complete, open the new account from the **All Resources** tile.



Add a graph

You can now use Data Explorer to create a graph container and add data to your database.

1. In the Azure portal, in the navigation menu, click **Data Explorer**.
2. In the Data Explorer blade, click **New Graph**, then fill in the page using the following information.



Add Graph

* Database id
Choose exisisting or type new id

* Graph Id
e.g., Collection1

* Storage capacity
Fixed(10 GB) | Unlimited

* Throughput (400 - 10,000 RU/s)
400

Partition key
e.g., /address/zipCode

SETTING	SUGGESTED VALUE	DESCRIPTION
Database id	sample-database	The ID for your new database. Database names must be between 1 and 255 characters, and cannot contain `/ \ # ?` or a trailing space.
Graph id	sample-graph	The ID for your new graph. Graph names have the same character requirements as database ids.
Storage Capacity	10 GB	Leave the default value. This is the storage capacity of the database.
Throughput	400 RUs	Leave the default value. You can scale up the throughput later if you want to reduce latency.
Partition key	/userid	A partition key that will distribute data evenly to each partition. Selecting the correct partition key is important in creating a performant graph, read more about it in Designing for partitioning.

3. Once the form is filled out, click **OK**.

Connect to your app service

1. Before starting the Gremlin Console, create or modify your *remote-secure.yaml* configuration file in your *apache-tinkerpop-gremlin-console-3.2.4/conf* directory.
2. Fill in your *host*, *port*, *username*, *password*, *connectionPool*, and *serializer* configurations:

SETTING	SUGGESTED VALUE	DESCRIPTION
Hosts	***.graphs.azure.com	Your graph service URI, which you can retrieve from the Azure portal
Port	443	Set to 443
Username	*Your username*	The resource of the form `/dbs/<db>/colls/<coll>`.
Password	*Your primary master key*	Your primary master key for the Azure Cosmos DB
ConnectionPool	{enableSsl: true}	Your connection pool setting for SSL
Serializer	{ className:org.apache.tinkerpop.gremlin. driver.ser.GraphSONMessageSerializerV1d0, config: { serializeResultToString: true }}	Set to this value

3. In your terminal, run *bin/gremlin.bat* or *bin/gremlin.sh* to start the Gremlin Console.

4. In your terminal, run *:remote connect tinkerpop.server conf/remote-secure.yaml* to connect to your app service.

Great! Now that we finished the setup, let's start running some console commands.

Let's try a simple count() command. Type the following in to the console at the prompt:

```
:> g.V().count()
```

> TIP
>
> Notice the **:>** that precedes the g.V().count() text?
>
> This is part of the command you need to type. It is important when using the Gremlin console, with Azure Cosmos DB.
>
> Omitting this :> prefix instructs the console to execute the command locally, often against an in-memory graph. Using this **:>** tells the console to execute a remote command, in this case against Cosmos DB (either the localhost emulator, or an > Azure instance).

Create vertices and edges

Let's begin by adding five person vertices for *Thomas*, *Mary Kay*, *Robin*, *Ben*, and *Jack*.

Input (Thomas):

```
:> g.addV('person').property('firstName', 'Thomas').property('lastName', 'Andersen').property('age', 44).property('userid', 1)
```

Output:

```
==>[id:796cdccc-2acd-4e58-a324-91d6f6f5ed6d,label:person,type:vertex,properties:[firstName:[[id:f02a749f-b67c-4016-850e-
910242d68953,value:Thomas]],lastName:[[id:f5fa3126-8818-4fda-88b0-9bb55145ce5c,value:Andersen]],age:[[id:f6390f9c-e563-433e-acbf-
25627628016e,value:44]],userid:[[id:796cdccc-2acd-4e58-a324-91d6f6f5ed6d|userid,value:1]]]]
```

Input (Mary Kay):

```
:> g.addV('person').property('firstName', 'Mary Kay').property('lastName', 'Andersen').property('age', 39).property('userid', 2)
```

Output:

```
==>[id:0ac9be25-a476-4a30-8da8-e79f0119ea5e,label:person,type:vertex,properties:[firstName:[[id:ea0604f8-14ee-4513-a48a-
1734a1f28dc0,value:Mary Kay]],lastName:[[id:86d3bba5-fd60-4856-9396-c195ef7d7f4b,value:Andersen]],age:[[id:bc81b78d-30c4-4e03-8f40-
50f72eb5f6da,value:39]],userid:[[id:0ac9be25-a476-4a30-8da8-e79f0119ea5e|userid,value:2]]]]
```

Input (Robin):

```
:> g.addV('person').property('firstName', 'Robin').property('lastName', 'Wakefield').property('userid', 3)
```

Output:

```
==>[id:8dc14d6a-8683-4a54-8d74-7eef1fb43a3e,label:person,type:vertex,properties:[firstName:[[id:ec65f078-7a43-4cbe-bc06-
e50f2640dc4e,value:Robin]],lastName:[[id:a3937d07-0e88-45d3-a442-26fcdfb042ce,value:Wakefield]],userid:[[id:8dc14d6a-8683-4a54-8d74-
7eef1fb43a3e|userid,value:3]]]]
```

Input (Ben):

```
:> g.addV('person').property('firstName', 'Ben').property('lastName', 'Miller').property('userid', 4)
```

Output:

```
==>[id:ee86b670-4d24-4966-9a39-30529284b66f,label:person,type:vertex,properties:[firstName:[[id:a632469b-30fc-4157-840c-
b80260871e9a,value:Ben]],lastName:[[id:4a08d307-0719-47c6-84ae-1b0b06630928,value:Miller]],userid:[[id:ee86b670-4d24-4966-9a39-
30529284b66f|userid,value:4]]]]
```

Input (Jack):

```
:> g.addV('person').property('firstName', 'Jack').property('lastName', 'Connor').property('userid', 5)
```

Output:

```
==>[id:4c835f2a-ea5b-43bb-9b6b-215488ad8469,label:person,type:vertex,properties:[firstName:[[id:4250824e-4b72-417f-af98-
8034aa15559f,value:Jack]],lastName:[[id:44c1d5e1-a831-480a-bf94-5167d133549e,value:Connor]],userid:[[id:4c835f2a-ea5b-43bb-9b6b-
215488ad8469|userid,value:5]]]]
```

Next, let's add edges for relationships between our people.

Input (Thomas -> Mary Kay):

```
:> g.V().hasLabel('person').has('firstName', 'Thomas').addE('knows').to(g.V().hasLabel('person').has('firstName', 'Mary Kay'))
```

Output:

```
==>[id:c12bf9fb-96a1-4cb7-a3f8-431e196e702f,label:knows,type:edge,inVLabel:person,outVLabel:person,inV:0d1fa428-780c-49a5-bd3a-
a68d96391d5c,outV:1ce821c6-aa3d-4170-a0b7-d14d2a4d18c3]
```

Input (Thomas -> Robin):

```
:> g.V().hasLabel('person').has('firstName', 'Thomas').addE('knows').to(g.V().hasLabel('person').has('firstName', 'Robin'))
```

Output:

```
==>[id:58319bdd-1d3e-4f17-a106-0ddf18719d15,label:knows,type:edge,inVLabel:person,outVLabel:person,inV:3e324073-ccfc-4ae1-8675-
d450858ca116,outV:1ce821c6-aa3d-4170-a0b7-d14d2a4d18c3]
```

Input (Robin -> Ben):

```
:> g.V().hasLabel('person').has('firstName', 'Robin').addE('knows').to(g.V().hasLabel('person').has('firstName', 'Ben'))
```

Output:

```
==>[id:889c4d3c-549e-4d35-bc21-a3d1bfa11e00,label:knows,type:edge,inVLabel:person,outVLabel:person,inV:40fd641d-546e-412a-abcc-
58fe53891aab,outV:3e324073-ccfc-4ae1-8675-d450858ca116]
```

Update a vertex

Let's update the *Thomas* vertex with a new age of *45*.

Input:

```
:> g.V().hasLabel('person').has('firstName', 'Thomas').property('age', 45)
```

Output:

```
==>[id:ae36f938-210e-445a-92df-519f2b64c8ec,label:person,type:vertex,properties:[firstName:[[id:872090b6-6a77-456a-9a55-
a59141d4ebc2,value:Thomas]],lastName:[[id:7ee7a39a-a414-4127-89b4-870bc4ef99f3,value:Andersen]],age:[[id:a2a75d5a-ae70-4095-806d-
a35abcbfe71d,value:45]]]]
```

Query your graph

Now, let's run a variety of queries against your graph.

First, let's try a query with a filter to return only people who are older than 40 years old.

Input (filter query):

```
:> g.V().hasLabel('person').has('age', gt(40))
```

Output:

```
==>[id:ae36f938-210e-445a-92df-519f2b64c8ec,label:person,type:vertex,properties:[firstName:[[id:872090b6-6a77-456a-9a55-
a59141d4ebc2,value:Thomas]],lastName:[[id:7ee7a39a-a414-4127-89b4-870bc4ef99f3,value:Andersen]],age:[[id:a2a75d5a-ae70-4095-806d-
a35abcbfe71d,value:45]]]]
```

Next, let's project the first name for the people who are older than 40 years old.

Input (filter + projection query):

```
:> g.V().hasLabel('person').has('age', gt(40)).values('firstName')
```

Output:

```
==>Thomas
```

Traverse your graph

Let's traverse the graph to return all of Thomas's friends.

Input (friends of Thomas):

```
:> g.V().hasLabel('person').has('firstName', 'Thomas').outE('knows').inV().hasLabel('person')
```

Output:

```
==>[id:f04bc00b-cb56-46c4-a3bb-a5870c42f7ff,label:person,type:vertex,properties:[firstName:[[id:14feedec-b070-444e-b544-62be15c7167c,value:Mary Kay]],lastName:[[id:107ab421-7208-45d4-b969-bbc54481992a,value:Andersen]],age:[[id:4b08d6e4-58f5-45df-8e69-6b790b692e0a,value:39]]]]
==>[id:91605c63-4988-4b60-9a30-5144719ae326,label:person,type:vertex,properties:[firstName:[[id:f760e0e6-652a-481a-92b0-1767d9bf372e,value:Robin]],lastName:[[id:352a4caa-bad6-47e3-a7dc-90ff342cf870,value:Wakefield]]]]
```

Next, let's get the next layer of vertices. Traverse the graph to return all the friends of Thomas's friends.

Input (friends of friends of Thomas):

```
:> g.V().hasLabel('person').has('firstName', 'Thomas').outE('knows').inV().hasLabel('person').outE('knows').inV().hasLabel('person')
```

Output:

```
==>[id:a801a0cb-ee85-44ee-a502-271685ef212e,label:person,type:vertex,properties:[firstName:[[id:b9489902-d29a-4673-8c09-c2b3fe7f8b94,value:Ben]],lastName:[[id:e084f933-9a4b-4dbc-8273-f0171265cf1d,value:Miller]]]]
```

Drop a vertex

Let's now delete a vertex from the graph database.

Input (drop Jack vertex):

```
:> g.V().hasLabel('person').has('firstName', 'Jack').drop()
```

Clear your graph

Finally, let's clear the database of all vertices and edges.

Input:

```
:> g.E().drop()
:> g.V().drop()
```

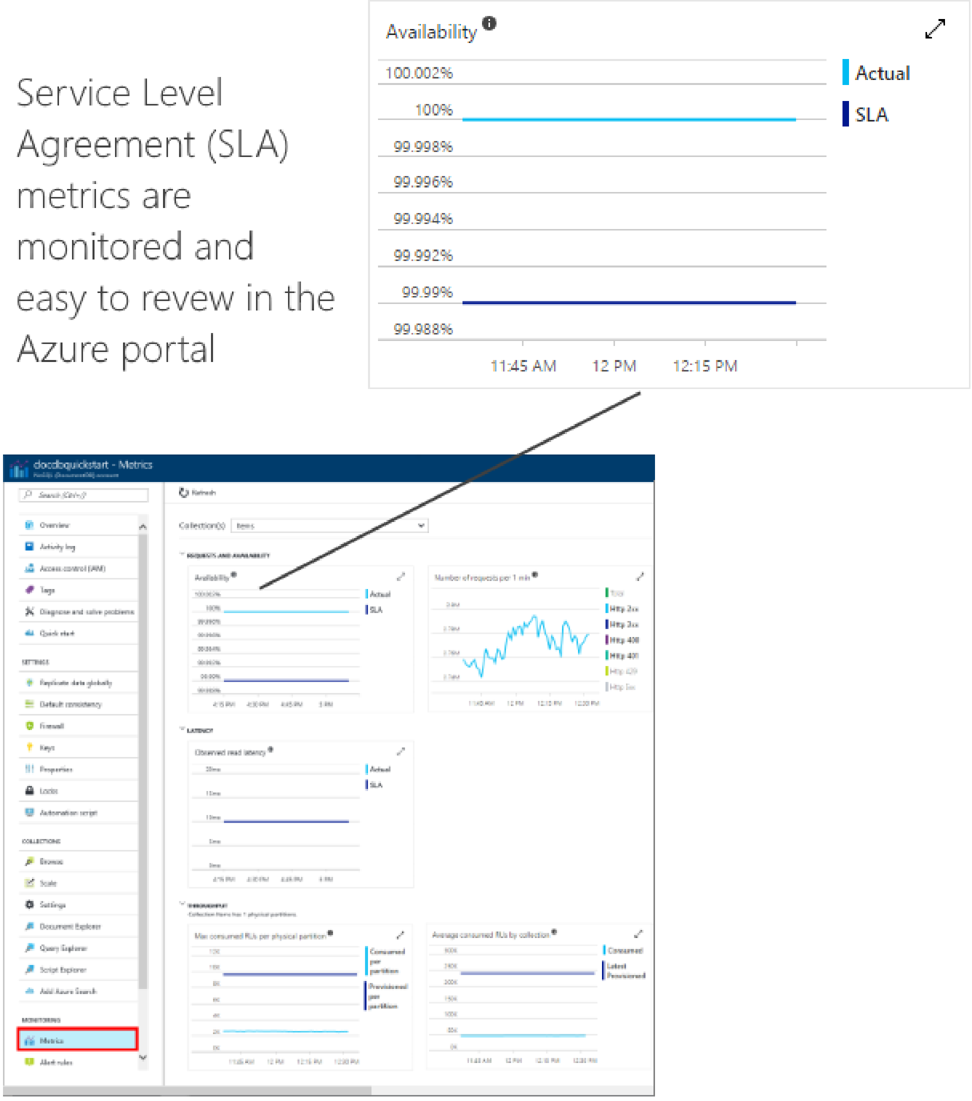
Congratulations! You've completed this Azure Cosmos DB: Graph API tutorial!

Review SLAs in the Azure portal

Now that your app is up and running, you'll want to ensure business continuity and watch user access to ensure high availability. You can use the Azure portal to review the availability, latency, throughput, and consistency of your collection.

Each graph that's associated with the Azure Cosmos DB Service Level Agreements (SLAs) provides a line that shows the quota required to meet the SLA and your actual usage, giving you a clear view into your database performance. Additional metrics, such as storage usage and number of requests per minute, are also included in the portal.

- In the Azure portal, in the left pane, under **Monitoring**, click **Metrics**.



Service Level Agreement (SLA) metrics are monitored and easy to revew in the Azure portal

Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart in the Azure portal with the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a graph using the Data Explorer, create vertices and edges, and traverse your graph using the Gremlin console. You can now build more complex queries and implement powerful graph traversal logic using Gremlin.

Query using Gremlin

Azure Cosmos DB: Build a Java application using the Graph API

6/6/2017 • 6 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quick start demonstrates how to create an Azure Cosmos DB account for Graph API (preview), database, and graph using the Azure portal. You then build and run a console app using the OSS Gremlin Java driver.

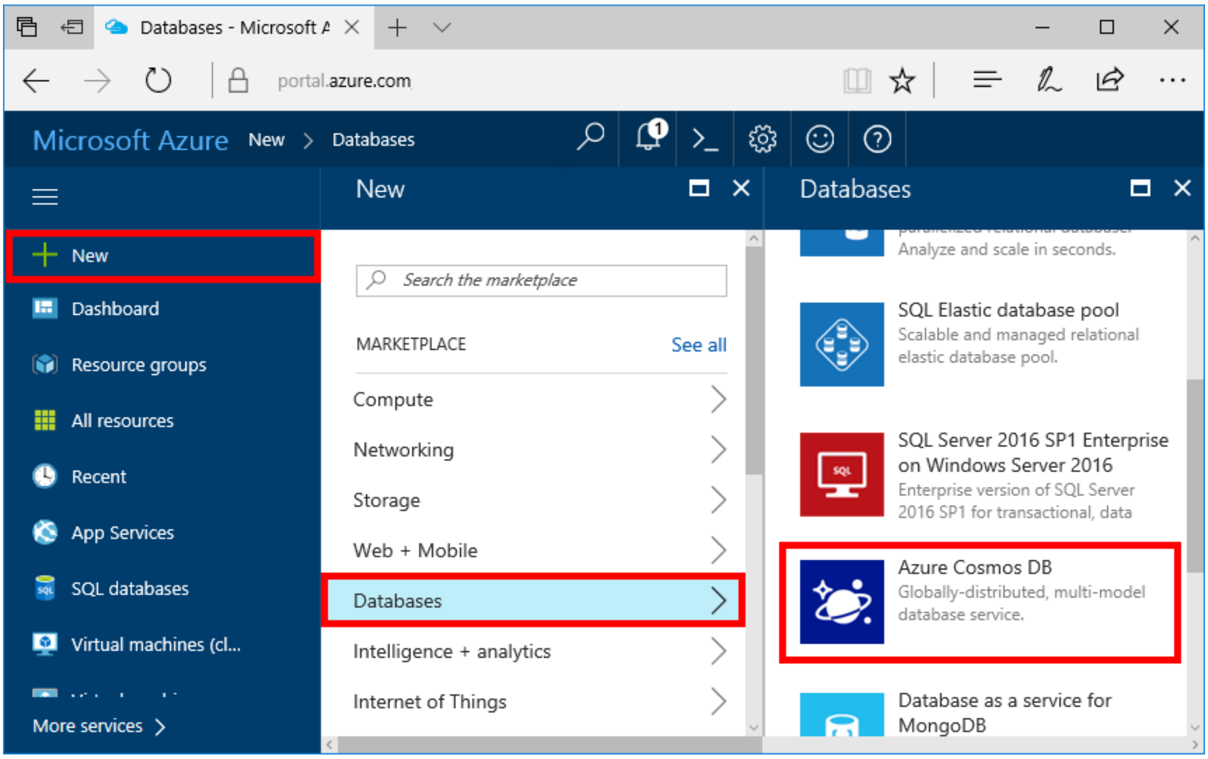
Prerequisites

- Before you can run this sample, you must have the following prerequisites:
  - JDK 1.7+ (Run `apt-get install default-jdk` if you don't have JDK), and set environment variables like `JAVA_HOME`
  - Maven (Run `apt-get install maven` if you don't have Maven)

If you don't have an Azure subscription, create a free account before you begin.

Create a database account

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

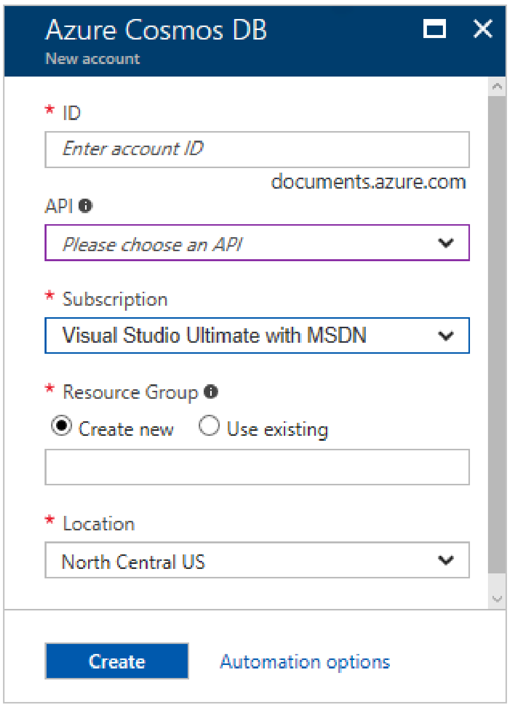


3. In the **New account** blade, specify the desired configuration for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

In this quick-start article, we program against the Graph API, so choose **Gremlin (graph)** as you fill out the form. If you have document data from a catalog app, key/value (table) data, or data that's migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

On the **New account** blade, complete the fields with the information in the following screenshot as a guide only. Because your own values will not match those in the screenshot, be sure to choose unique values as you set up your account.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

* Subscription
Visual Studio Ultimate with MSDN

* Resource Group
● Create new ○ Use existing

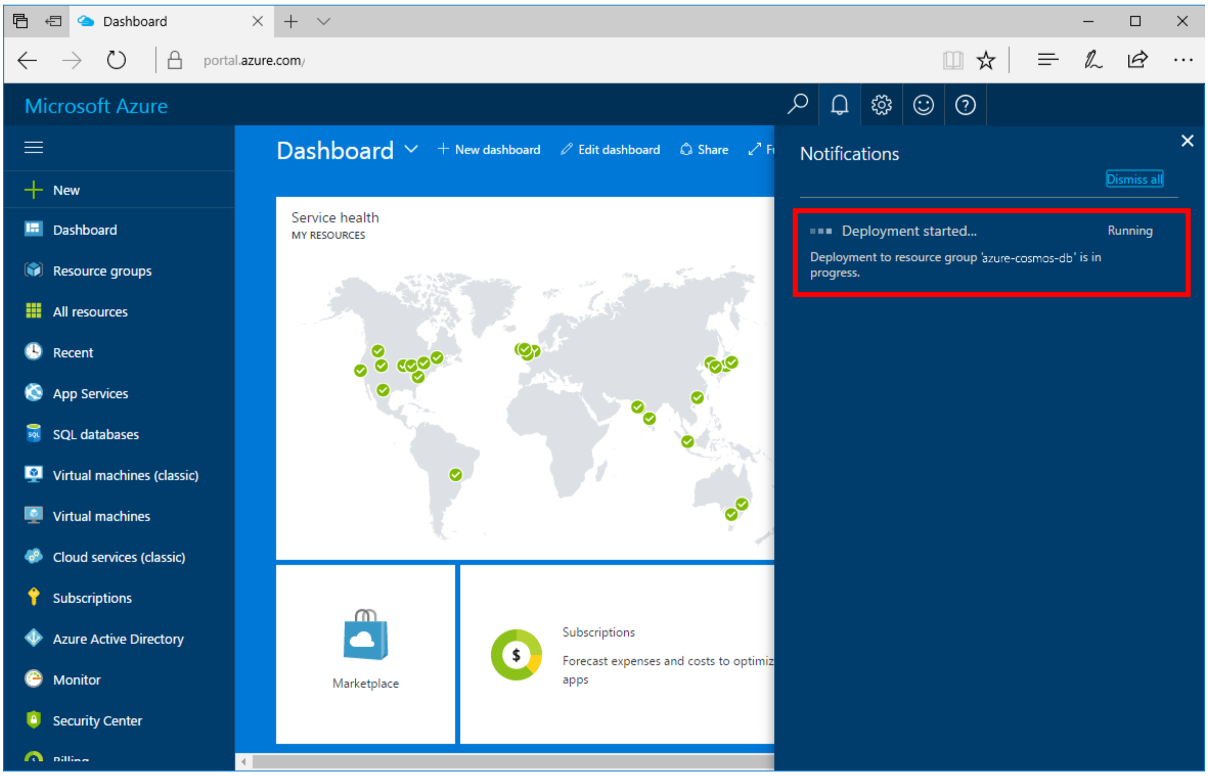
* Location
North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that you choose to identify the Azure Cosmos DB account. Because *documents.azure.com* is appended to the ID that you provide to create your URI, use a unique but identifiable ID. The ID must contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 to 50 characters.
API	Gremlin (graph)	We program against the Graph API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for the Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location closest to your users to give them the fastest access to the data.

SETTING	SUGGESTED VALUE	DESCRIPTION

4. Click **Create** to create the account.

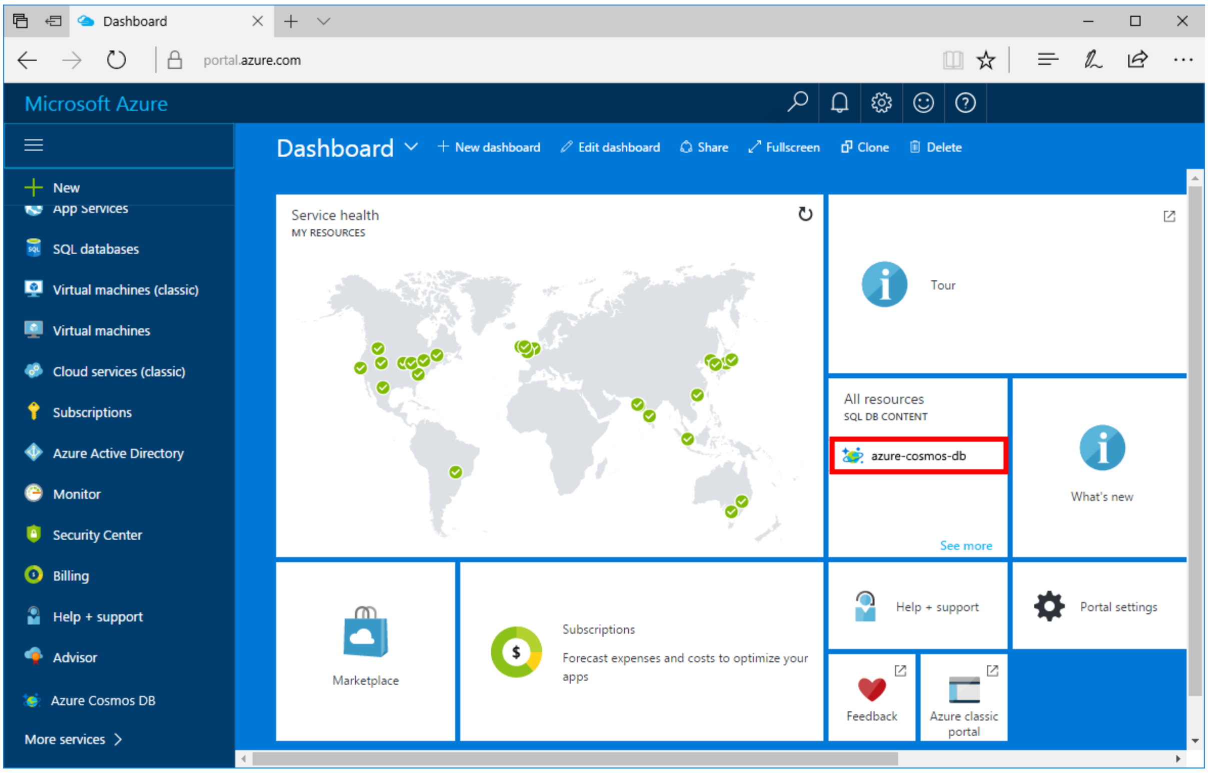
5. On the toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

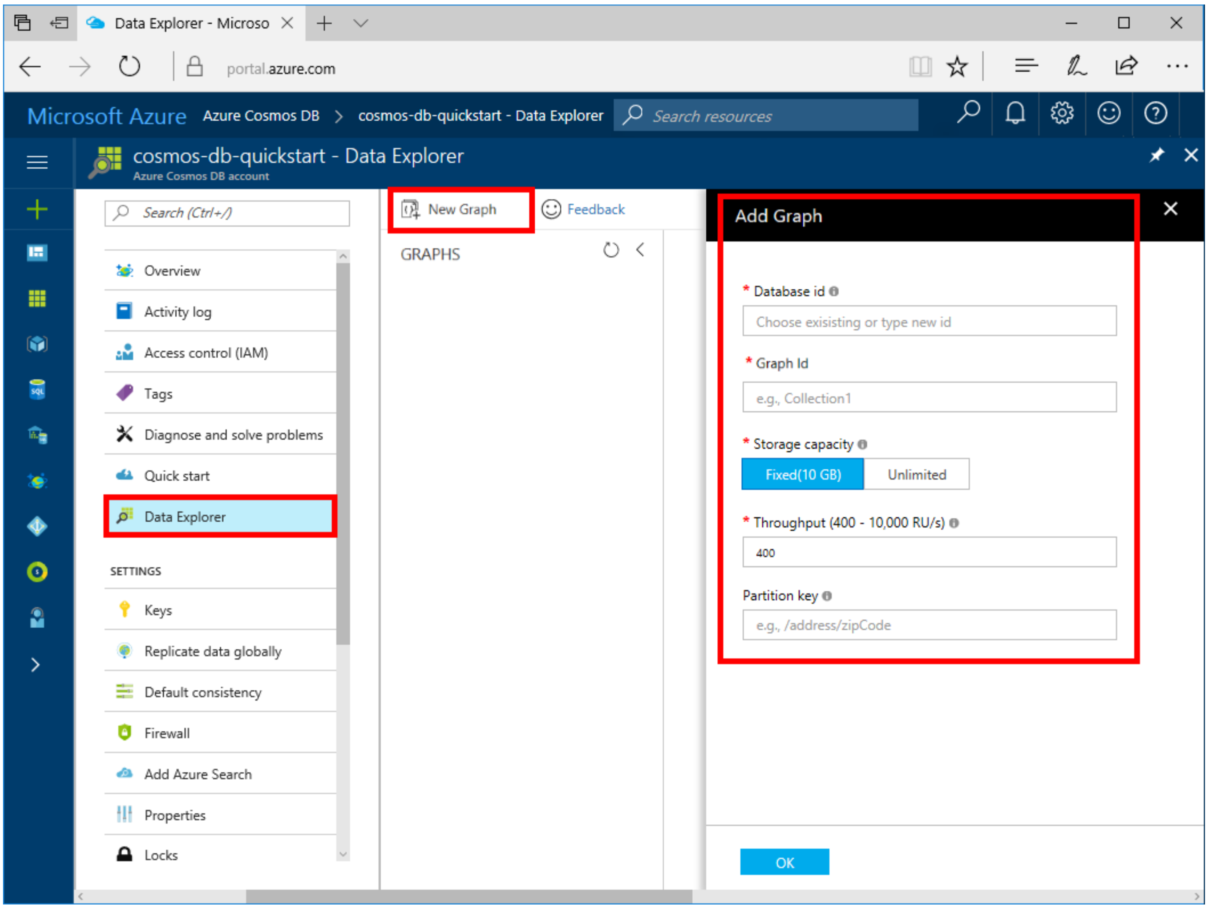
6. When the deployment is complete, open the new account from the **All Resources** tile.



Add a graph

You can now use Data Explorer to create a graph container and add data to your database.

1. In the Azure portal, in the navigation menu, click **Data Explorer**.
2. In the Data Explorer blade, click **New Graph**, then fill in the page using the following information.



* Database id: Choose exisisting or type new id

* Graph Id: e.g., Collection1

* Storage capacity: Fixed(10 GB) | Unlimited

* Throughput (400 - 10,000 RU/s): 400

Partition key: e.g., /address/zipCode

OK

SETTING	SUGGESTED VALUE	DESCRIPTION
Database id	sample-database	The ID for your new database. Database names must be between 1 and 255 characters, and cannot contain `/ \ # ?` or a trailing space.
Graph id	sample-graph	The ID for your new graph. Graph names have the same character requirements as database ids.
Storage Capacity	10 GB	Leave the default value. This is the storage capacity of the database.
Throughput	400 RUs	Leave the default value. You can scale up the throughput later if you want to reduce latency.
Partition key	/userid	A partition key that will distribute data evenly to each partition. Selecting the correct partition key is important in creating a performant graph, read more about it in Designing for partitioning.

3. Once the form is filled out, click **OK**.

Clone the sample application

Now let's clone a Graph API (preview) app from github, set the connection string, and run it. You see how easy it is to work with data programmatically.

1. Open a git terminal window, such as git bash, and `cd` to a working directory.

2. Run the following command to clone the sample repository.

   ```
   git clone https://github.com/Azure-Samples/azure-cosmos-db-graph-java-getting-started.git
   ```

Review the code

Let's make a quick review of what's happening in the app. Open the `Program.java` file and you find that these lines of code.

- The Gremlin `Client` is initialized from the configuration in `src/remote-secure.yaml` that you set earlier.

  ```
  Cluster cluster = Cluster.build(new File("src/remote.yaml")).create();

  Client client = cluster.connect();
  ```

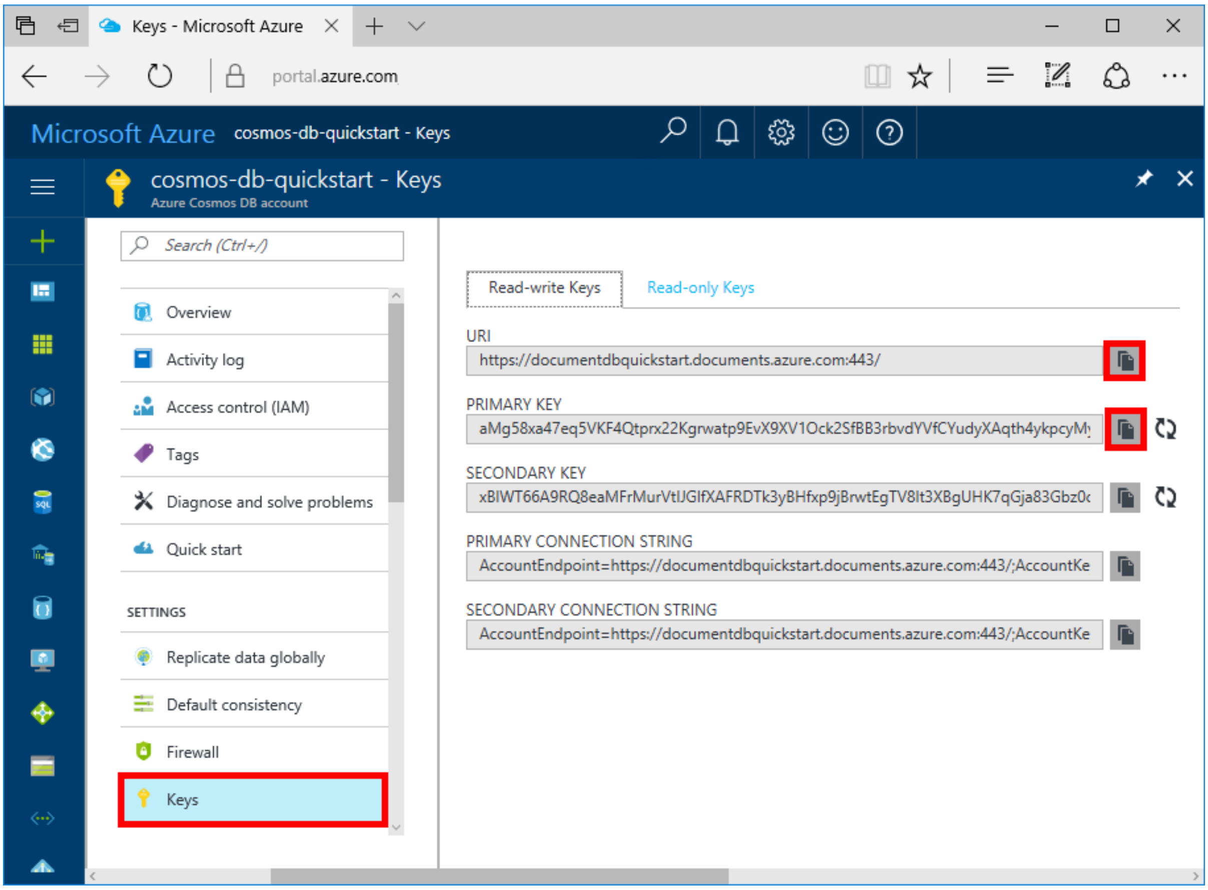
- A series of Gremlin steps are executed using the `client.submit` method.

  ```
  ResultSet results = client.submit("g.V()");

  CompletableFuture<List<Result>> completableFutureResults = results.all();
  List<Result> resultList = completableFutureResults.get();

  for (Result result : resultList) {
     System.out.println(result.toString());
  }
  ```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. In the Azure portal, in your Azure Cosmos DB account, in the left navigation click **Keys**, and then click **Read-write Keys**. You use the copy buttons on the right side of the screen to copy the URI and Primary Key into the `Program.java` file in the next step.



URI
https://documentdbquickstart.documents.azure.com:443/

PRIMARY KEY
aMg58xa47eq5VKF4Qtprx22Kgrwatp9EvX9XV1Ock2SfBB3rbvdYVfCYudyXAqth4ykpcyM

SECONDARY KEY
xBIWT66A9RQ8eaMFrMurVtIJGIfXAFRDTk3yBHfxp9jBrwtEgTV8It3XBgUHK7qGja83Gbz0

PRIMARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

SECONDARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

2. In Open the `src/remote-secure.yaml` file.

3. Fill in your *host*, *port*, *username*, *password*, *connectionPool*, and *serializer* configurations in the `src/remote-secure.yaml` file:

SETTING	SUGGESTED VALUE	DESCRIPTION
Hosts	***.graphs.azure.com	Your graph service URI, which you can retrieve from the Azure portal
Port	443	Set to 443
Username	*Your username*	The resource of the form `/dbs/<db>/colls/<coll>`.
Password	*Your primary master key*	Your primary master key for the Azure Cosmos DB
ConnectionPool	{enableSsl: true}	Your connection pool setting for SSL
Serializer	{ className:org.apache.tinkerpop.gremlin. driver.ser.GraphSONMessageSerializerV1d0, config: { serializeResultToString: true }}	Set to this value

Run the console app

1. Run `mvn package` in a terminal to install required npm modules

2. Run `mvn exec:java -D exec.mainClass=GetStarted.Program` in a terminal to start your Java application.

You can now go back to Data Explorer and see query, modify, and work with this new data.

Browse using the Data Explorer

You can now go back to Data Explorer in the Azure portal and browse and query your new graph data.

- In Data Explorer, the new database appears in the Collections pane. Expand **graphdb**, **graphcoll**, and then click **Graph**.

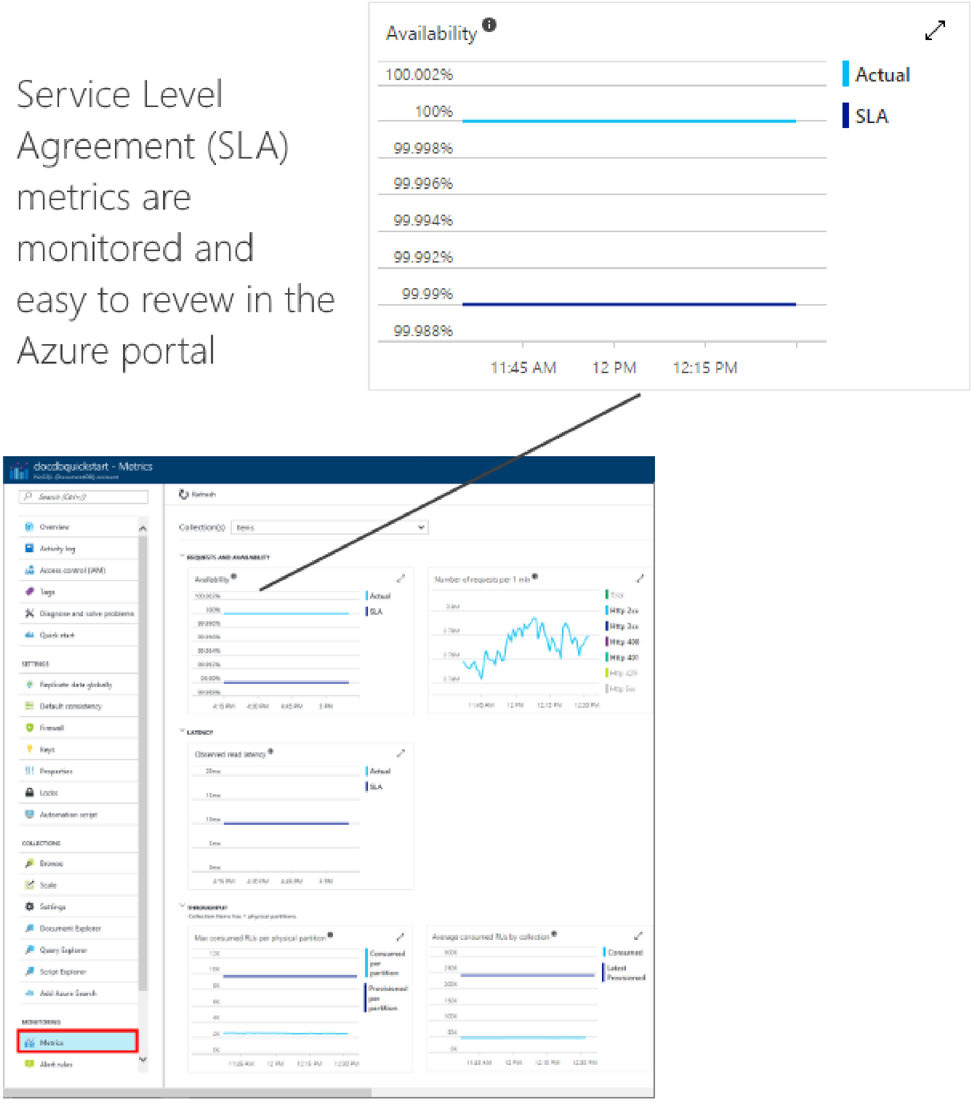
  The data generated by the sample app is displayed in the Graphs pane.

Review SLAs in the Azure portal

Now that your app is up and running, you'll want to ensure business continuity and watch user access to ensure high availability. You can use the Azure portal to review the availability, latency, throughput, and consistency of your collection.

Each graph that's associated with the Azure Cosmos DB Service Level Agreements (SLAs) provides a line that shows the quota required to meet the SLA and your actual usage, giving you a clear view into your database performance. Additional metrics, such as storage usage and number of requests per minute, are also included in the portal.

- In the Azure portal, in the left pane, under **Monitoring**, click **Metrics**.



Service Level Agreement (SLA) metrics are monitored and easy to revew in the Azure portal

Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart in the Azure portal with the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a graph using the Data Explorer, and run an app. You can now build more complex queries and implement powerful graph traversal logic using Gremlin.

Query using Gremlin

Azure Cosmos DB: Build a Node.js application by using Graph API

6/6/2017 • 6 min to read • Edit Online

Azure Cosmos DB is the globally distributed multi-model database service from Microsoft. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quick-start article demonstrates how to create an Azure Cosmos DB account for Graph API (preview), database, and graph by using the Azure portal. You then build and run a console app by using the open-source Gremlin Node.js driver.

> NOTE
>
> The npm module `gremlin-secure` is a modified version of `gremlin` module, with support for SSL and SASL required for connecting with Azure Cosmos DB. Source code is available on GitHub.

Prerequisites

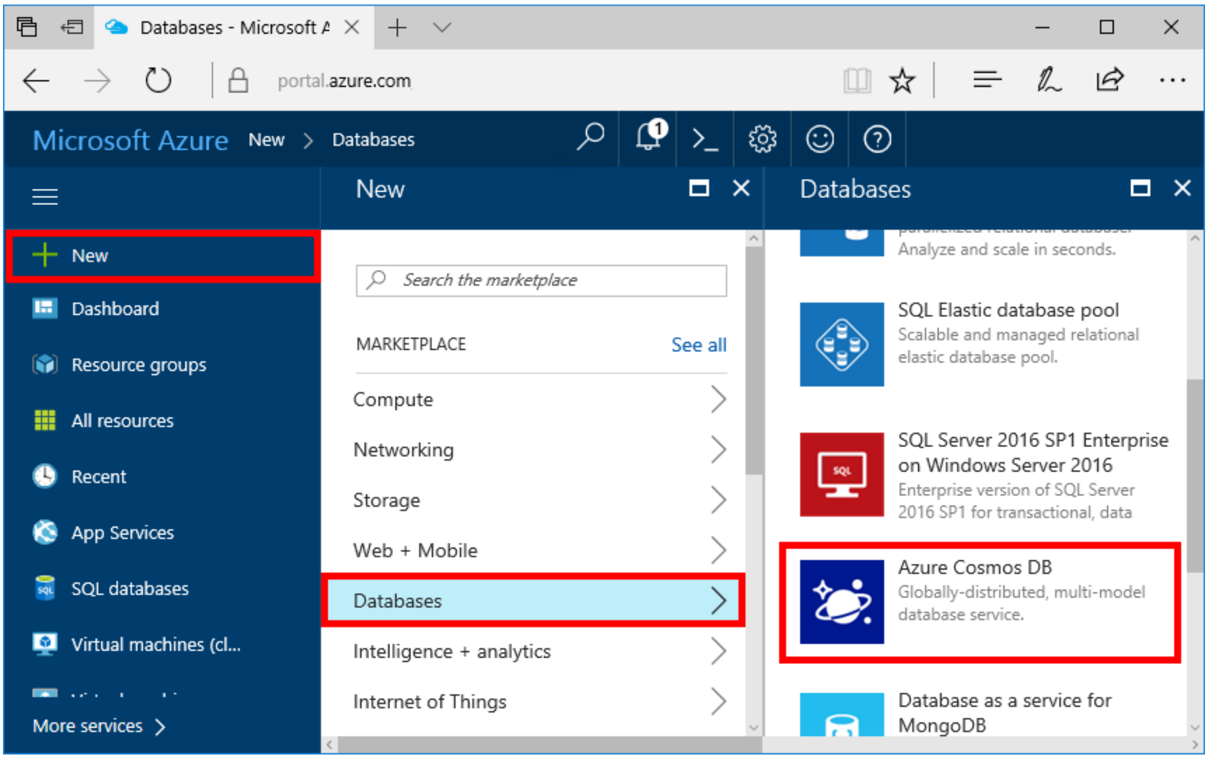
Before you can run this sample, you must have the following prerequisites:

- Node.js version v0.10.29 or later
- Git

If you don't have an Azure subscription, create a free account before you begin.

Create a database account

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

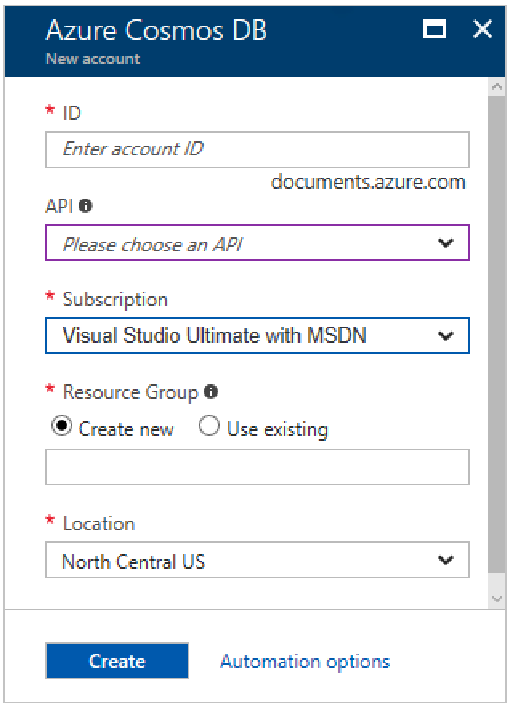


3. In the **New account** blade, specify the desired configuration for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article, we program against the Graph API, so choose **Gremlin (graph)** as you fill out the form. If you have document data from a catalog app, key/value (table) data, or data that's migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

   On the **New account** blade, complete the fields with the information in the following screenshot as a guide only. Because your own values will not match those in the screenshot, be sure to choose unique values as you set up your account.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

* Subscription
Visual Studio Ultimate with MSDN

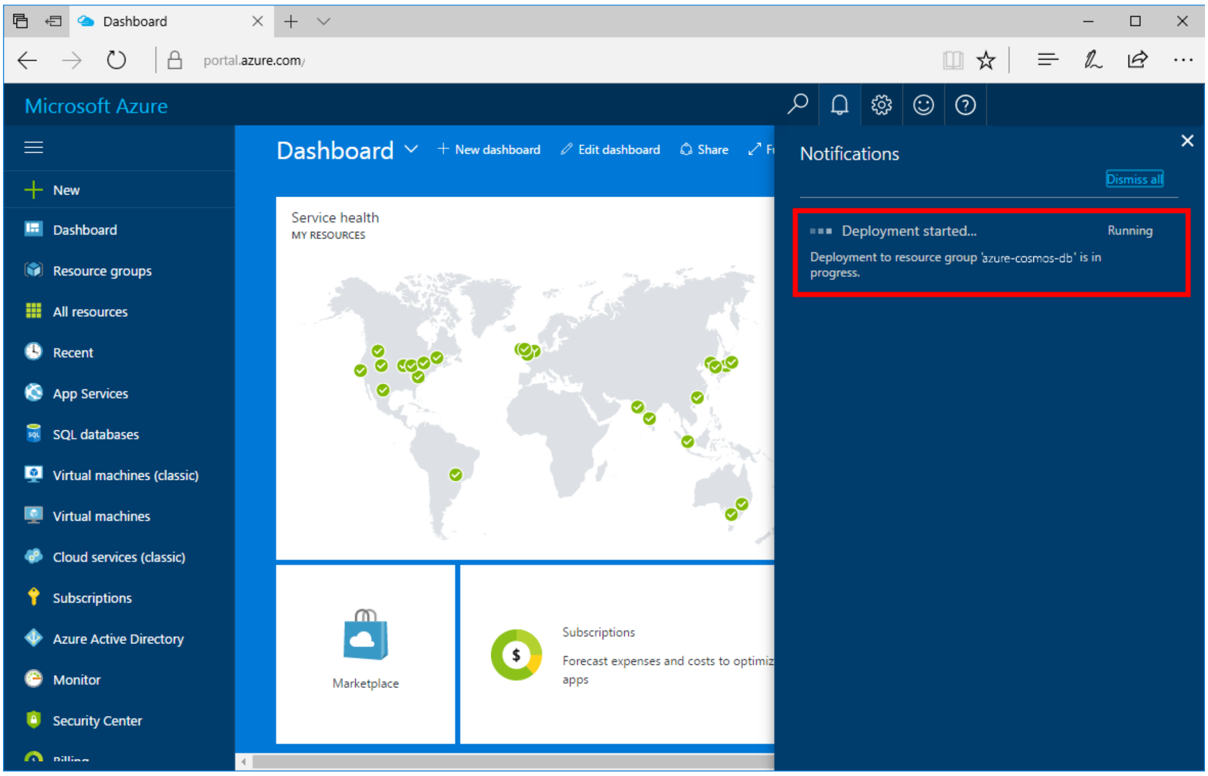
* Resource Group
● Create new ○ Use existing

* Location
North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that you choose to identify the Azure Cosmos DB account. Because *documents.azure.com* is appended to the ID that you provide to create your URI, use a unique but identifiable ID. The ID must contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 to 50 characters.
API	Gremlin (graph)	We program against the Graph API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for the Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location closest to your users to give them the fastest access to the data.

4. Click **Create** to create the account.

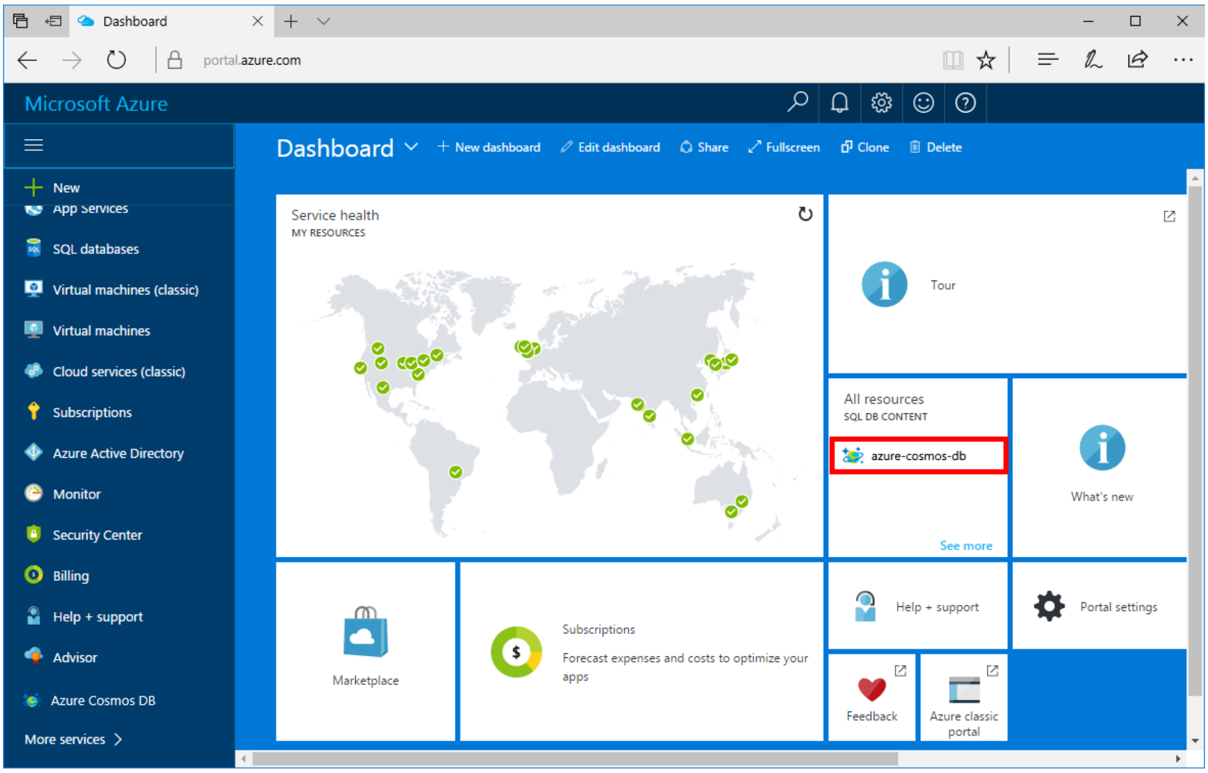
5. On the toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

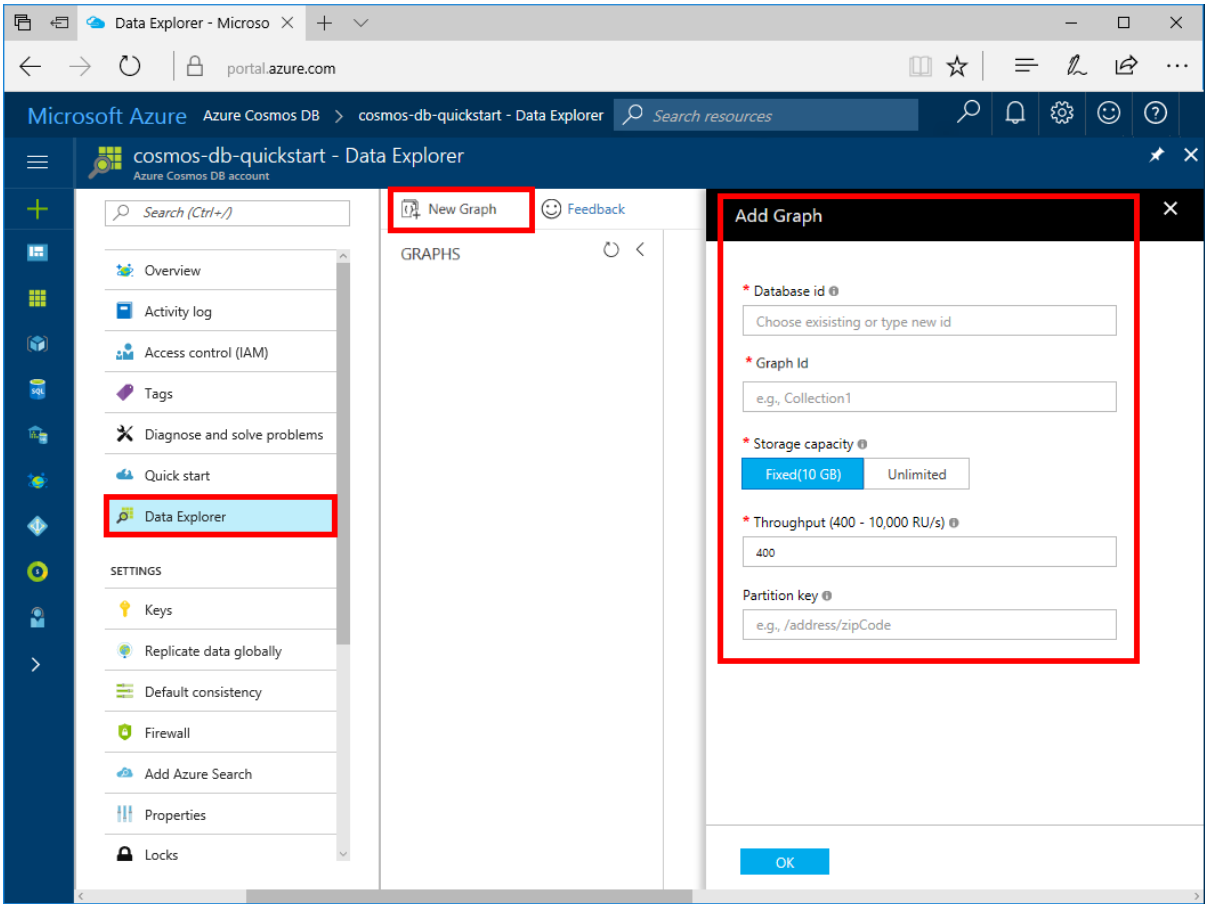
6. When the deployment is complete, open the new account from the **All Resources** tile.



Add a graph

You can now use Data Explorer to create a graph container and add data to your database.

1. In the Azure portal, in the navigation menu, click **Data Explorer**.
2. In the Data Explorer blade, click **New Graph**, then fill in the page using the following information.



Add Graph

* Database id
Choose exisisting or type new id

* Graph Id
e.g., Collection1

* Storage capacity
Fixed(10 GB) | Unlimited

* Throughput (400 - 10,000 RU/s)
400

Partition key
e.g., /address/zipCode

SETTING	SUGGESTED VALUE	DESCRIPTION
Database id	sample-database	The ID for your new database. Database names must be between 1 and 255 characters, and cannot contain `/ \ # ?` or a trailing space.
Graph id	sample-graph	The ID for your new graph. Graph names have the same character requirements as database ids.
Storage Capacity	10 GB	Leave the default value. This is the storage capacity of the database.
Throughput	400 RUs	Leave the default value. You can scale up the throughput later if you want to reduce latency.
Partition key	/userid	A partition key that will distribute data evenly to each partition. Selecting the correct partition key is important in creating a performant graph, read more about it in Designing for partitioning.

3. Once the form is filled out, click **OK**.

Clone the sample application

Now let's clone a Graph API app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a Git terminal window, such as Git Bash, and change (via `cd` command) to a working directory.

2. Run the following command to clone the sample repository.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-graph-nodejs-getting-started.git
```

3. Open the solution file in Visual Studio.

Review the code

Let's make a quick review of what's happening in the app. Open the `app.js` file, and you'll find the following lines of code.

- The Gremlin client is created.

```
const client = Gremlin.createClient(
  443,
  config.endpoint,
  {
    "session": false,
    "ssl": true,
    "user": `/dbs/${config.database}/colls/${config.collection}`,
    "password": config.primaryKey
  });
```

The configurations are all in `config.js`, which we edit in the following section.

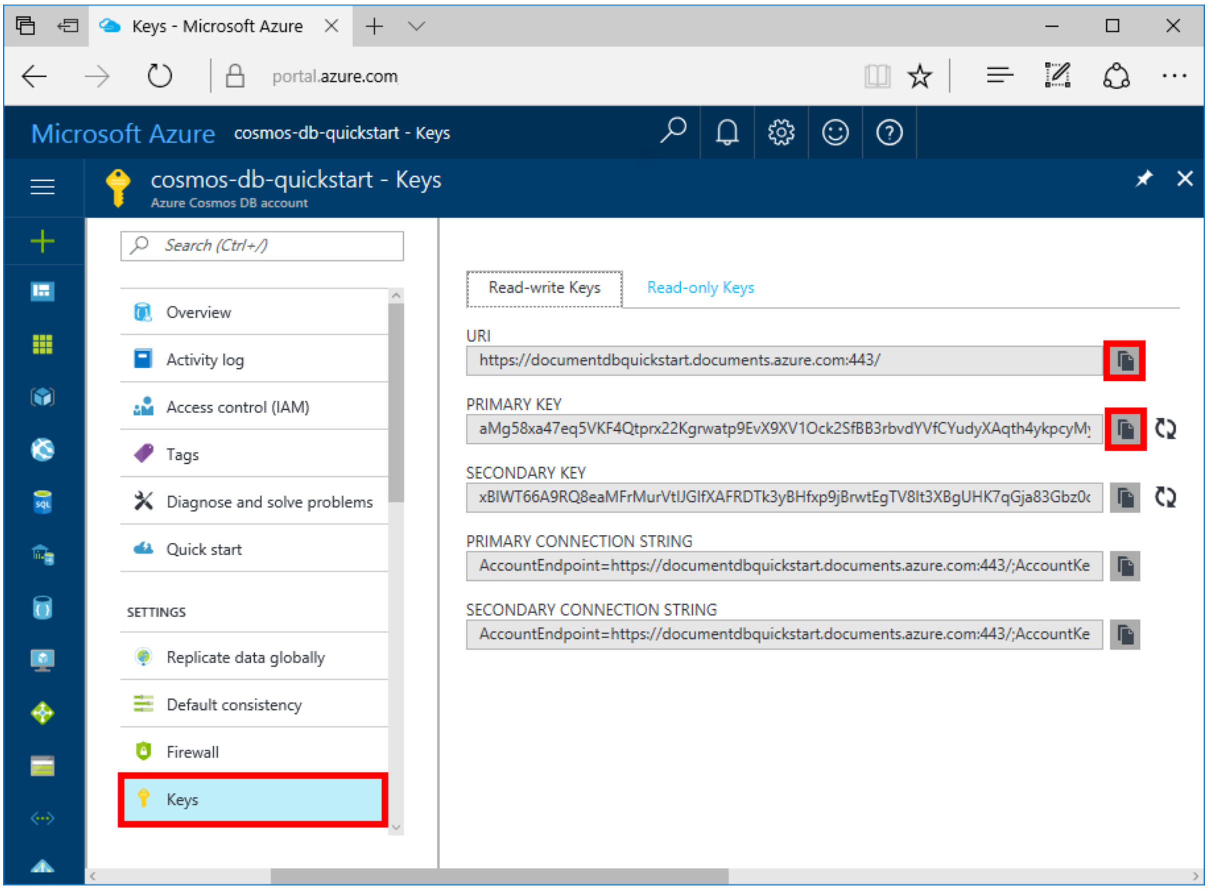
- A series of Gremlin steps are executed with the `client.execute` method.

```
console.log('Running Count');
client.execute("g.V().count()", { }, (err, results) => {
  if (err) return console.error(err);
  console.log(JSON.stringify(results));
  console.log();
});
```

Update your connection string

Now go back to the Azure portal to get your connection string information, and copy it into the app.

1. In the Azure portal, in your Azure Cosmos DB account, on the left navigation menu, click **Keys**, and then click **Read-write Keys**. You use the copy buttons at the right to copy the URI and primary key into the `app.js` file in the next step.



URI
https://documentdbquickstart.documents.azure.com:443/

PRIMARY KEY
aMg58xa47eq5VKF4Qtprx22Kgrwatp9EvX9XV1Ock2SfBB3rbvdYVfCYudyXAqth4ykpcyMy

SECONDARY KEY
xBIWT66A9RQ8eaMFrMurVtIJGIfXAFRDTk3yBHfxp9jBrwtEgTV8It3XBgUHK7qGja83Gbz0c

PRIMARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

SECONDARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

2. Copy your Gremlin URI value from the portal (using the copy button) and make it the value of `config.endpoint` key in config.js. The Gremlin endpoint must be only the host name without the protocol/port number, like `mygraphdb.graphs.azure.com` (not `https://mygraphdb.graphs.azure.com` or `mygraphdb.graphs.azure.com:433`).

   ```
   config.endpoint = "GRAPHENDPOINT";
   ```

3. Copy your primary key value from the portal and make it the value of config.primaryKey in config.js. You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

   ```
   config.primaryKey = "PRIMARYKEY";
   ```

4. Enter the database name, and graph (container) name for the value of config.database and config.collection.

Here is an example of what your completed config.js file should look like:

```
var config = {}

// Note that this must not have HTTPS or the port number
config.endpoint = "mygraphdb.graphs.azure.com";
config.primaryKey = "OjlhK6tjxfSXyKtrmCiM9O6gQQgu5DmgAoauzD1PdPIq1LZJmILTarHvrolyUYOB0whGQ4j21rdAFwoYep7Kkw==";
config.database = "graphdb"
config.collection = "Persons"

module.exports = config;
```

Run the console app

1. Open a terminal window and change (via `cd` command) to the installation directory for the package.json file that's included in the project.

2. Run `npm install` to install the required npm modules, including `gremlin-secure`.

3. Run `node app.js` in a terminal to start your node application.

Browse with Data Explorer

You can now go back to Data Explorer in the Azure portal to view, query, modify, and work with your new graph data.

In Data Explorer, the new database appears in the **Collections** pane. Expand **graphdb**, **graphcoll**, and then click **Graph**.

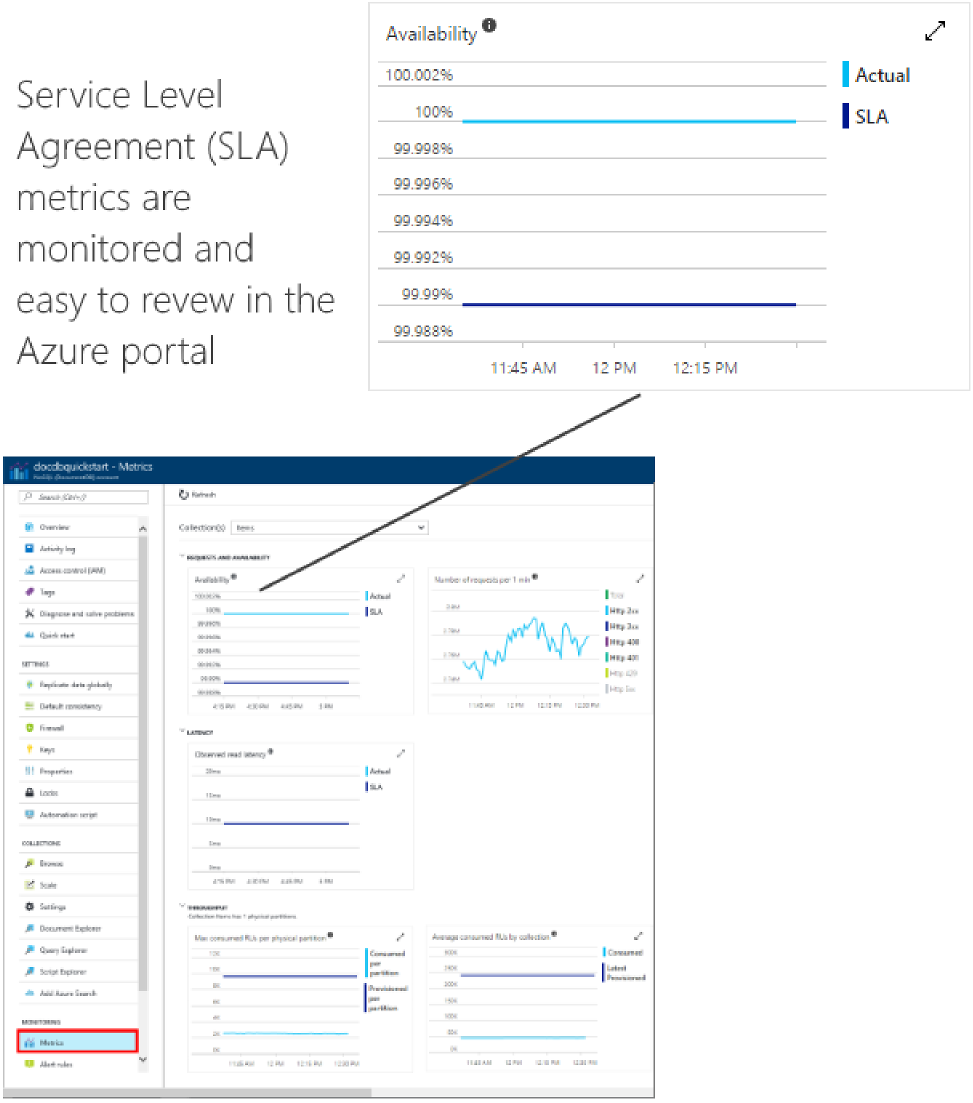
The data generated by the sample app is displayed in the **Graphs** pane.

Review SLAs in the Azure portal

Now that your app is up and running, you'll want to ensure business continuity and watch user access to ensure high availability. You can use the Azure portal to review the availability, latency, throughput, and consistency of your collection.

Each graph that's associated with the Azure Cosmos DB Service Level Agreements (SLAs) provides a line that shows the quota required to meet the SLA and your actual usage, giving you a clear view into your database performance. Additional metrics, such as storage usage and number of requests per minute, are also included in the portal.

- In the Azure portal, in the left pane, under **Monitoring**, click **Metrics**.



Service Level Agreement (SLA) metrics are monitored and easy to revew in the Azure portal

Clean up your resources

If you do not plan to continue using this app, delete all resources that you created in this article by doing the following:

1. In the Azure portal, on the left navigation menu, click **Resource groups**, and then click the name of the resource that you created.
2. On your resource group page, click **Delete**, type the name of the resource to be deleted, and then click **Delete**.

Next steps

In this article, you've learned how to create an Azure Cosmos DB account, create a graph by using Data Explorer, and run an app. You can now build more complex queries and implement powerful graph traversal logic by using Gremlin.

Query using Gremlin

Azure Cosmos DB: Build a .NET application using the Table API

6/7/2017 • 7 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quick start demonstrates how to create an Azure Cosmos DB account, and create a table within that account using the Azure portal. You'll then write code to insert, update, and delete entities, and run some queries using the new Windows Azure Storage Premium Table (preview) package from NuGet. This library has the same classes and method signatures as the public Windows Azure Storage SDK, but also has the ability to connect to Azure Cosmos DB accounts using the Table API (preview).

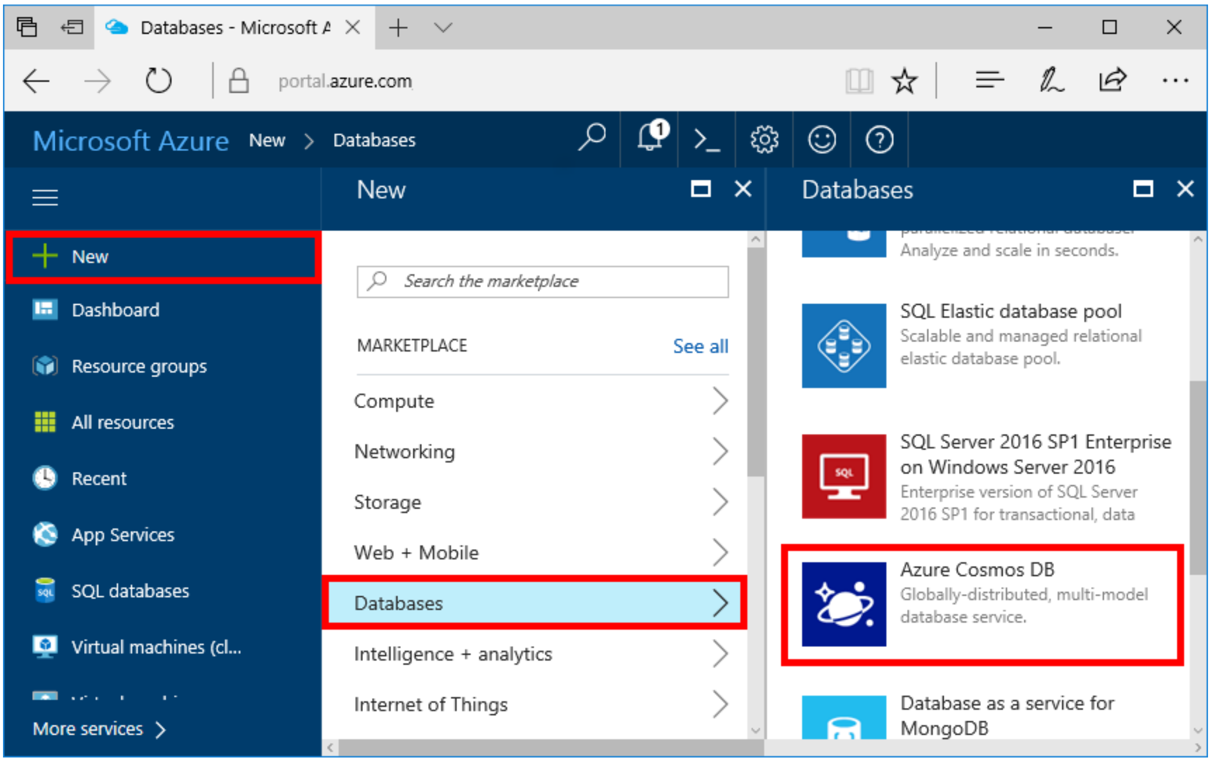
Prerequisites

If you don't already have Visual Studio 2017 installed, you can download and use the **free** Visual Studio 2017 Community Edition. Make sure that you enable **Azure development** during the Visual Studio setup.

If you don't have an Azure subscription, create a free account before you begin.

Create a database account

1. In a new window, sign in to the Azure portal.
2. In the left menu, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

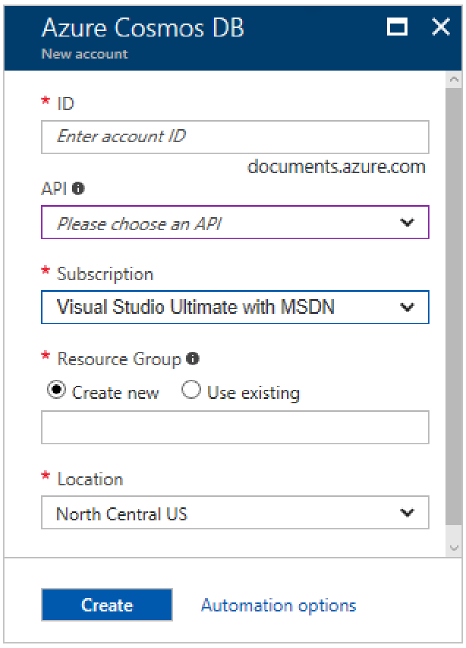


3. In the **New account** blade, specify the desired configuration for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick start we'll be programming against the Table API so you'll choose **Table (key-value)** as you fill out the form. But if you have graph data for a social media app, document data from a catalog app, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally-distributed database service platform for all your mission-critical applications.

Fill out the New account blade using the information in the screenshot as a guide. You will choose unique values as you set up your account so your values will not match the screenshot exactly.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

* Subscription
Visual Studio Ultimate with MSDN

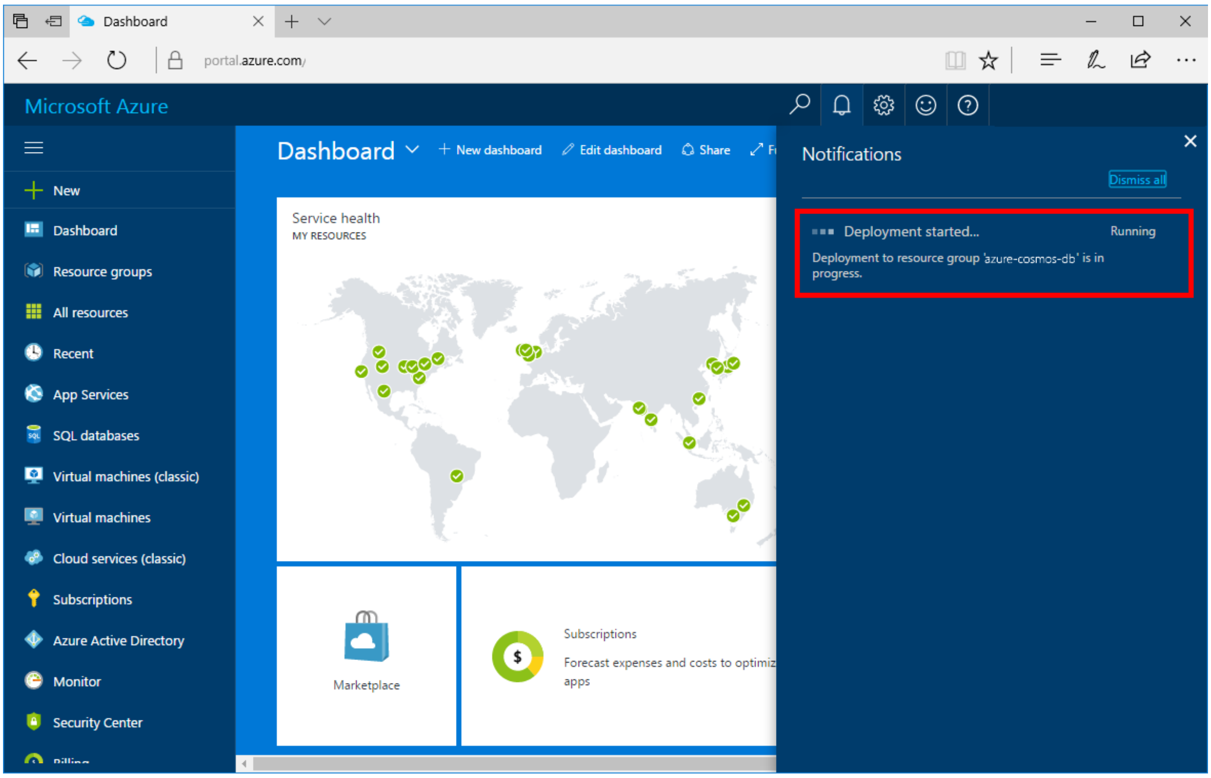
* Resource Group
● Create new ○ Use existing

* Location
North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name you choose to identify the Azure Cosmos DB account. *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID may contain only lowercase letters, numbers, and the '-' character, and must be between 3 and 50 characters.
API	Table (key-value)	We'll be programming against the Table API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for the Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location closest to your users to give them the fastest access to the data.

4. Click **Create** to create the account.

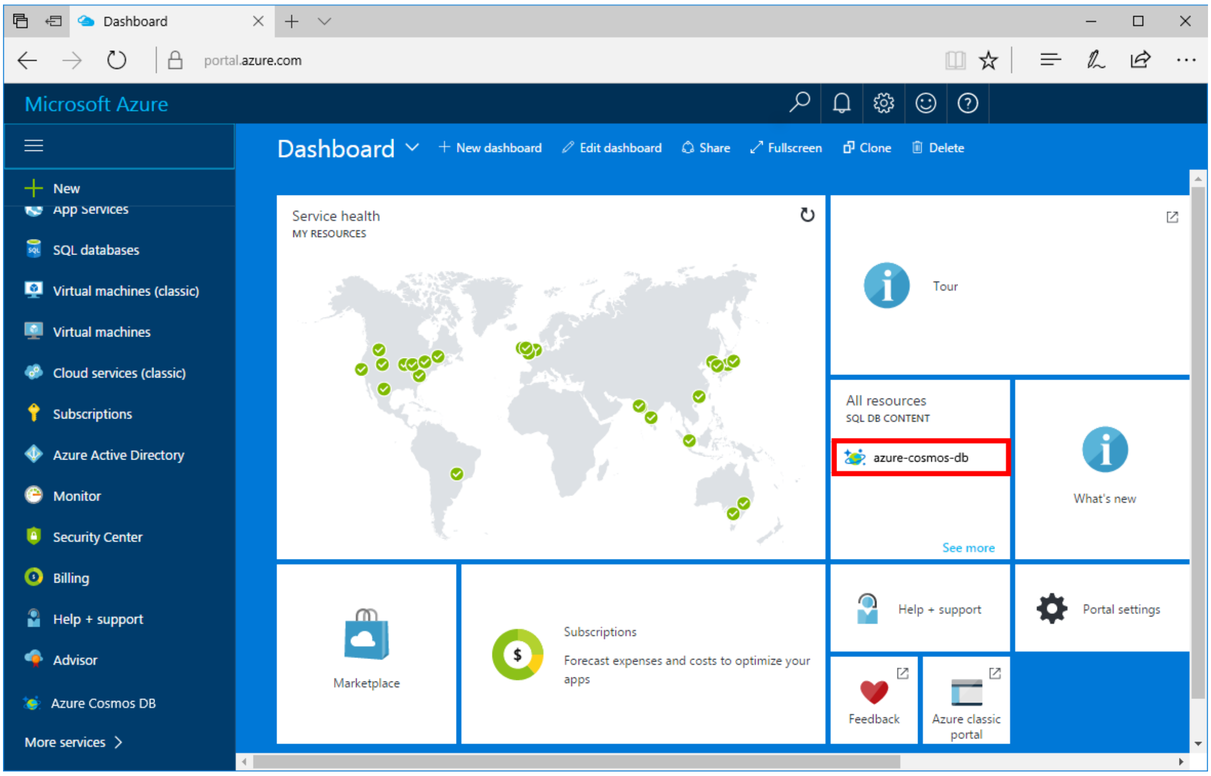
5. On the toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

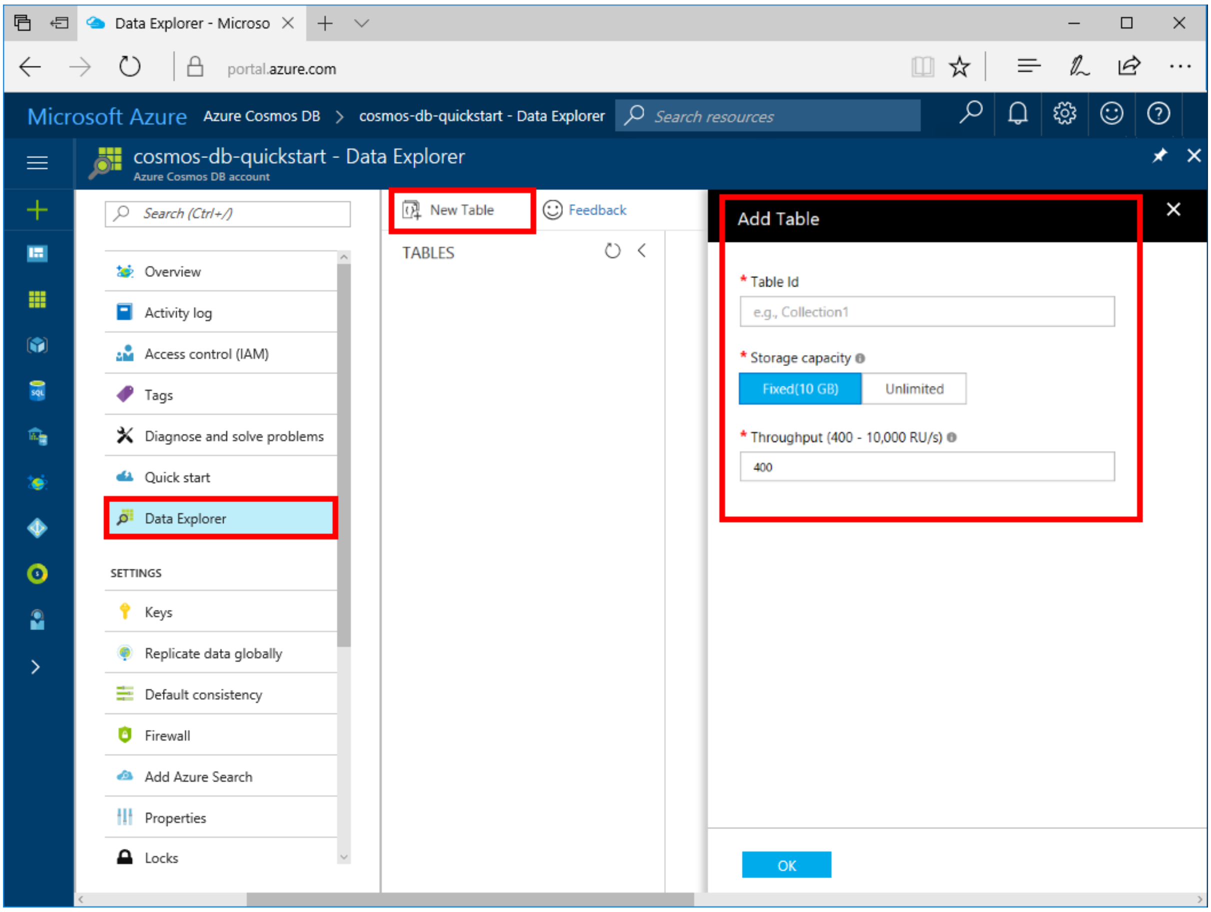
6. When the deployment is complete, open the new account from the All Resources tile.



Add a table

You can now use Data Explorer to create a graph container and add data to your database.

1. In the Azure portal, in the navigation menu, click **Data Explorer**.
2. In the Data Explorer blade, click **New Table**, then fill in the page using the following information.



Add Table

* Table Id

e.g., Collection1

* Storage capacity

Fixed(10 GB) | Unlimited

* Throughput (400 - 10,000 RU/s)

400

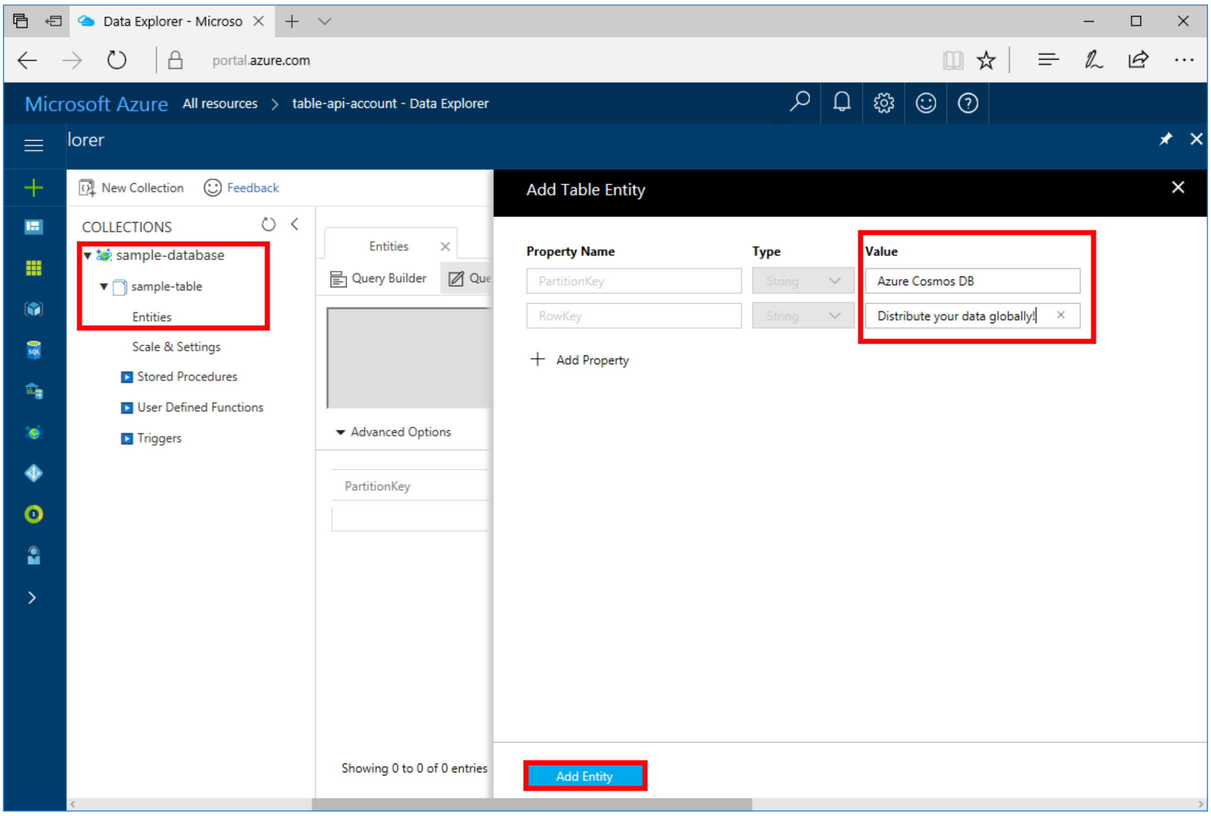
SETTING	SUGGESTED VALUE	DESCRIPTION
Database id	sample-database	The ID for your new database. Database names must be between 1 and 255 characters, and cannot contain `/ \ # ?` or a trailing space.
Table id	sample-table	The ID for your new table. Table names have the same character requirements as database ids.
Storage Capacity	10 GB	Leave the default value. This is the storage capacity of the database.
Throughput	400 RUs	Leave the default value. You can scale up the throughput later if you want to reduce latency.

3. Once the form is filled out, click **OK**.

Add sample data

You can now add data to your new table using Data Explorer.

1. In Data Explorer, expand **sample-database**, **sample-table**, click **Entities**, and then click **Add Entity**.
2. Now add data to the PartitionKey value box and RowKey value box, and click **Add Entity**.



You can now add more entities to your table, edit your entities, or query your data in Data Explorer. Data Explorer is also where you can scale your throughput and add stored procedures, user defined functions, and triggers to your table.

Clone the sample application

Now let's clone a DocumentDB API app from github, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a git terminal window, such as git bash, and `cd` to a working directory.
2. Run the following command to clone the sample repository.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-table-dotnet-getting-started.git
```

3. Then open the solution file in Visual Studio.

Review the code

Let's make a quick review of what's happening in the app. Open the Program.cs file and you'll find that these lines of code create the Azure Cosmos DB resources.

- The CloudTableClient is initialized.

```
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(connectionString);
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();
```

- A new table is created if it does not exist.

```
CloudTable table = tableClient.GetTableReference("people");
table.CreateIfNotExists();
```

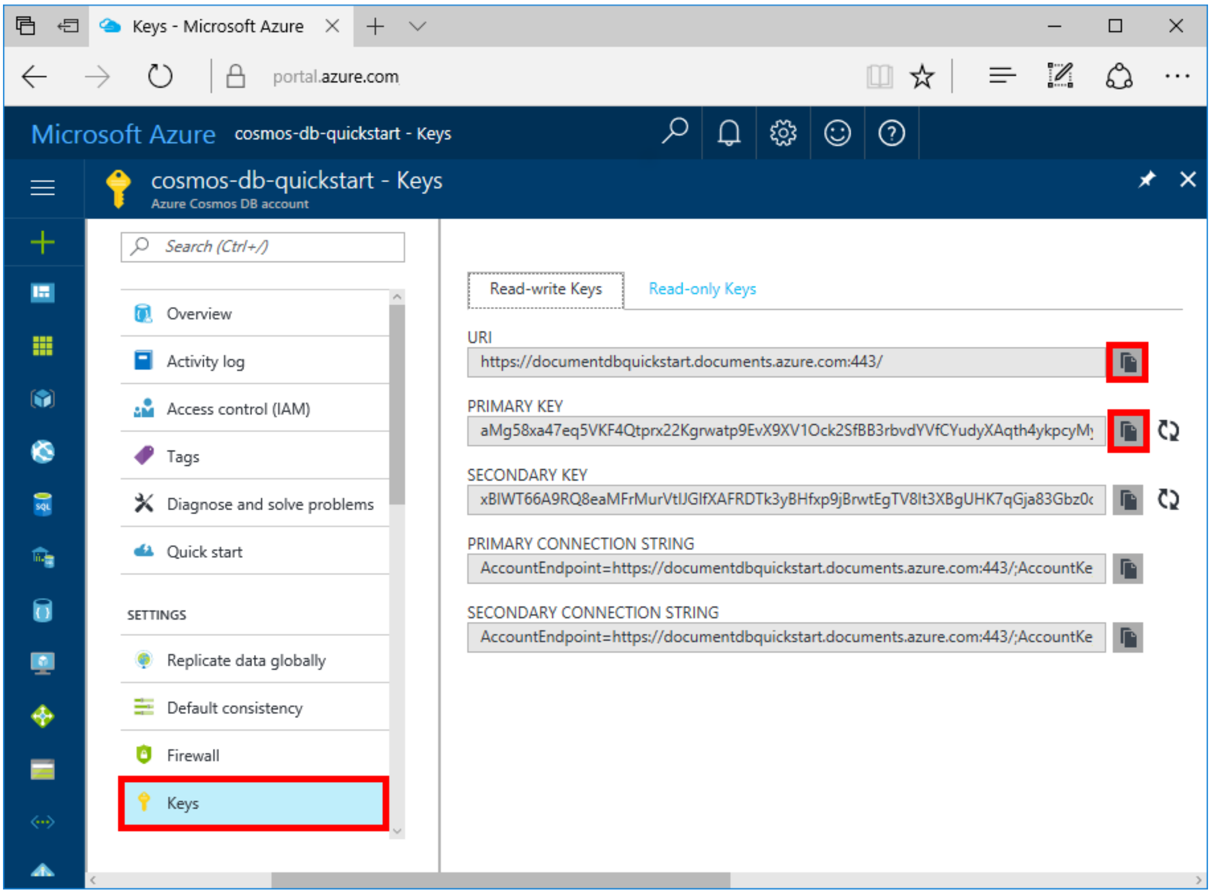
- A new Table container is created. You will notice this code very similar to regular Azure Table storage SDK

```
CustomerEntity item = new CustomerEntity()
        {
            PartitionKey = Guid.NewGuid().ToString(),
            RowKey = Guid.NewGuid().ToString(),
            Email = $"{GetRandomString(6)}@contoso.com",
            PhoneNumber = "425-555-0102",
            Bio = GetRandomString(1000)
        };
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. In the Azure portal, in your Azure Cosmos DB account, in the left navigation click **Keys**, and then click **Read-write Keys**. You'll use the copy buttons on the right side of the screen to copy the URI and Primary Key into the app.config file in the next step.



URI
https://documentdbquickstart.documents.azure.com:443/

PRIMARY KEY
aMg58xa47eq5VKF4Qtprx22Kgrwatp9EvX9XV1Ock2SfBB3rbvdYVfCYudyXAqth4ykpcyMy

SECONDARY KEY
xBIWT66A9RQ8eaMFrMurVtIJGIfXAFRDTk3yBHfxp9jBrwtEgTV8It3XBgUHK7qGja83Gbz0c

PRIMARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

SECONDARY CONNECTION STRING
AccountEndpoint=https://documentdbquickstart.documents.azure.com:443/;AccountKe

2. In Visual Studio, open the app.config file.

3. Copy your Azure Cosmos DB account name from the portal and make it the value of the AccountName in the PremiumStorageConnection string value in app.config. In the screenshot above, the account name is cosmos-db-quickstart. Your account name is in displayed at the top of the portal.

```
<add key="PremiumStorageConnectionString"
value="DefaultEndpointsProtocol=https;AccountName=MYSTORAGEACCOUNT;AccountKey=AUTHKEY;TableEndpoint=https://COSMOSDB.documents.azure.com"
/>
```

4. Then copy your PRIMARY KEY value from the portal and make it the value of the AccountKey in PremiumStorageConnectionString.

```
AccountKey=AUTHKEY
```

5. Finally, copy your URI value from the Keys page of the portal (using the copy button) and make it the value of the TableEndpoint of the PremiumStorageConnectionString.

```
TableEndpoint=https://COSMOSDB.documents.azure.com
```

You can leave the StandardStorageConnectionString as is.

You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

Run the web app

1. In Visual Studio, right-click on the project in **Solution Explorer** and then click **Manage NuGet Packages**.

2. In the NuGet **Browse** box, type *WindowsAzure.Storage* and check the **Include prerelease** box.

3. From the results, install the **Windows Azure Storage Premium Table** library. This installs the preview Azure Cosmos DB Table API package as well as all dependencies. Note that this is a different NuGet package than the Windows Azure Storage package used by Azure Table storage.

4. Click CTRL + F5 to run the application.

   The console window displays the data being added to the table. When the script completes, close the console window.

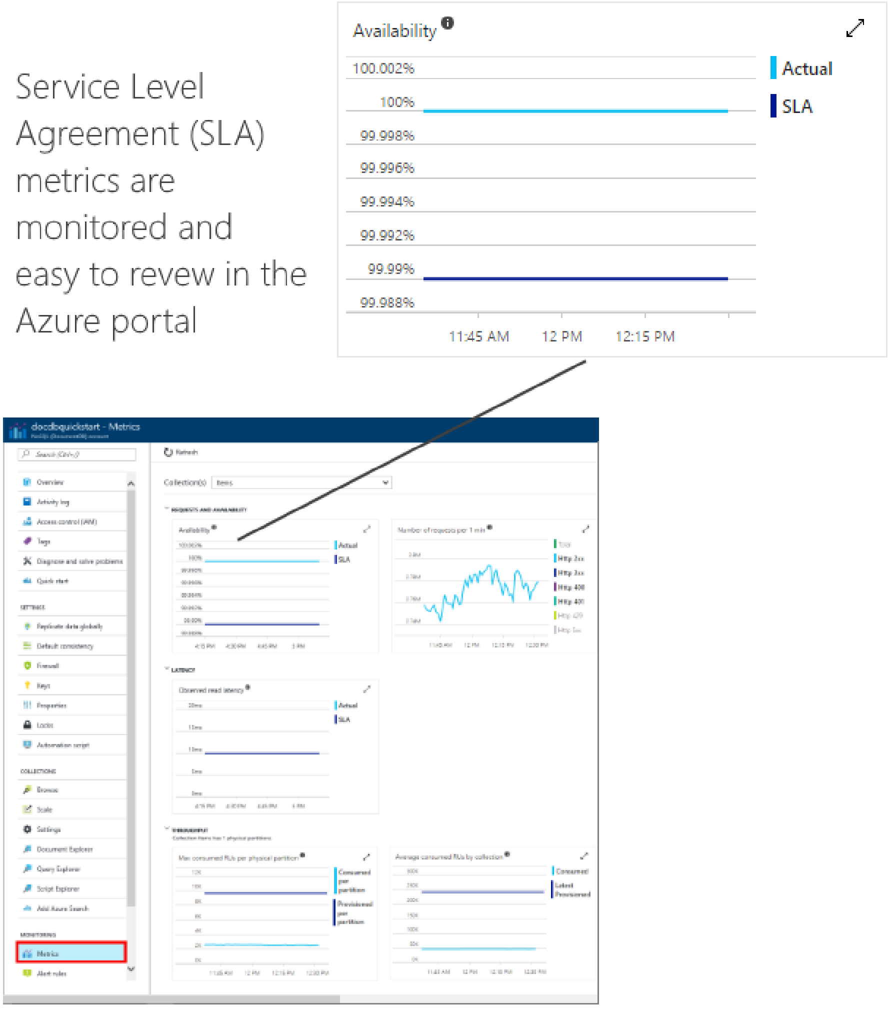
You can now go back to Data Explorer and see query, modify, and work with this new data.

Review SLAs in the Azure portal

Now that your app is up and running, you'll want to ensure business continuity and watch user access to ensure high availability. You can use the Azure portal to review the availability, latency, throughput, and consistency of your collection.

Each graph that's associated with the Azure Cosmos DB Service Level Agreements (SLAs) provides a line that shows the quota required to meet the SLA and your actual usage, giving you a clear view into your database performance. Additional metrics, such as storage usage and number of requests per minute, are also included in the portal.

- In the Azure portal, in the left pane, under **Monitoring**, click **Metrics**.



Service Level Agreement (SLA) metrics are monitored and easy to revew in the Azure portal

Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart in the Azure portal with the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a table using the Data Explorer, and run an app. Now you can query your data using the Table API.

Query using the Table API

Azure CosmosDB: Develop with the DocumentDB API in .NET

6/1/2017 • 11 min to read • Edit Online

Azure Cosmos DB is Microsoft’s globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This tutorial demonstrates how to create an Azure Cosmos DB account using the Azure portal, and then create a document database and collection with a partition key using the DocumentDB .NET API. By defining a partition key when you create a collection, your application is prepared to scale effortlessly as your data grows.

This tutorial covers the following tasks by using the DocumentDB .NET API:

- Create an Azure Cosmos DB account
- Create a database and collection with a partition key
- Create JSON documents
- Update a document
- Query partitioned collections
- Run stored procedures
- Delete a document
- Delete a database

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 Cosmos DB Emulator for this tutorial if you'd like to use a local environment that emulates the Azure DocumentDB service for development purposes.
- Visual Studio.

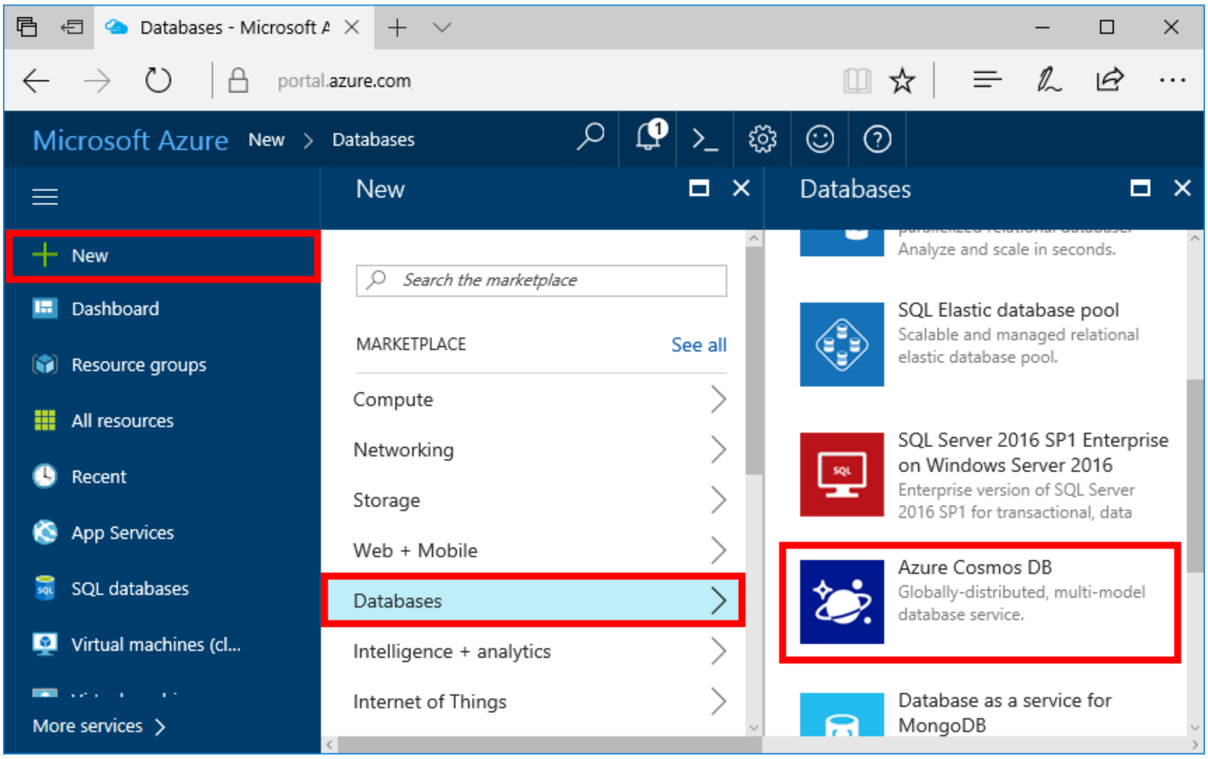
Create an Azure Cosmos DB account

Let's start by creating an Azure Cosmos DB account in the Azure portal.

TIP

- Already have an Azure Cosmos DB account? If so, skip ahead to Set up your Visual Studio solution
- Did you have an Azure DocumentDB account? If so, your account is now an Azure Cosmos DB account and you can skip ahead to Set up your Visual Studio solution.
- If you are using the Azure Cosmos DB Emulator, please follow the steps at Azure Cosmos DB Emulator to setup the emulator and skip ahead to Set up your Visual Studio Solution.

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

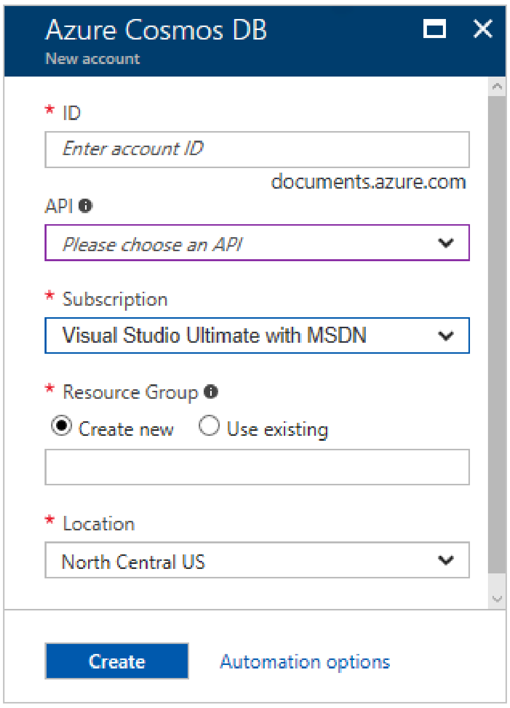


3. On the **New account** blade, specify the configuration that you want for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article we program against the DocumentDB API, so choose **SQL (DocumentDB)** as you fill out the form. But if you have graph data for a social media app, or key/value (table) data, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

   Complete the fields on the **New account** blade, using the information in the following screenshot as a guide. When you set up your account, choose unique values that do not match those in the screenshot.



Azure Cosmos DB
New account

* ID: Enter account ID .documents.azure.com

API: Please choose an API

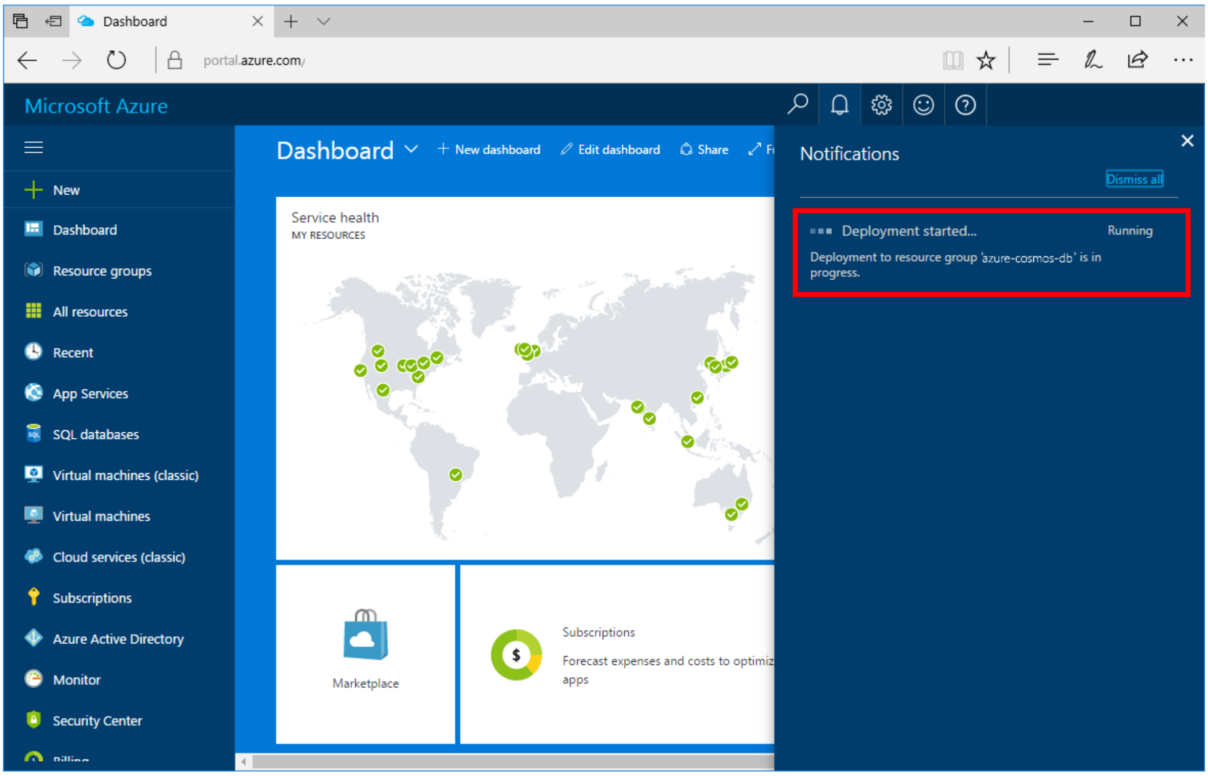
* Subscription: Visual Studio Ultimate with MSDN

* Resource Group: ● Create new ○ Use existing

* Location: North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that identifies your Azure Cosmos DB account. The string *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 through 50 characters.
API	SQL (DocumentDB)	We program against the DocumentDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for your Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource-group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location that's closest to your users to give them the fastest access to the data.

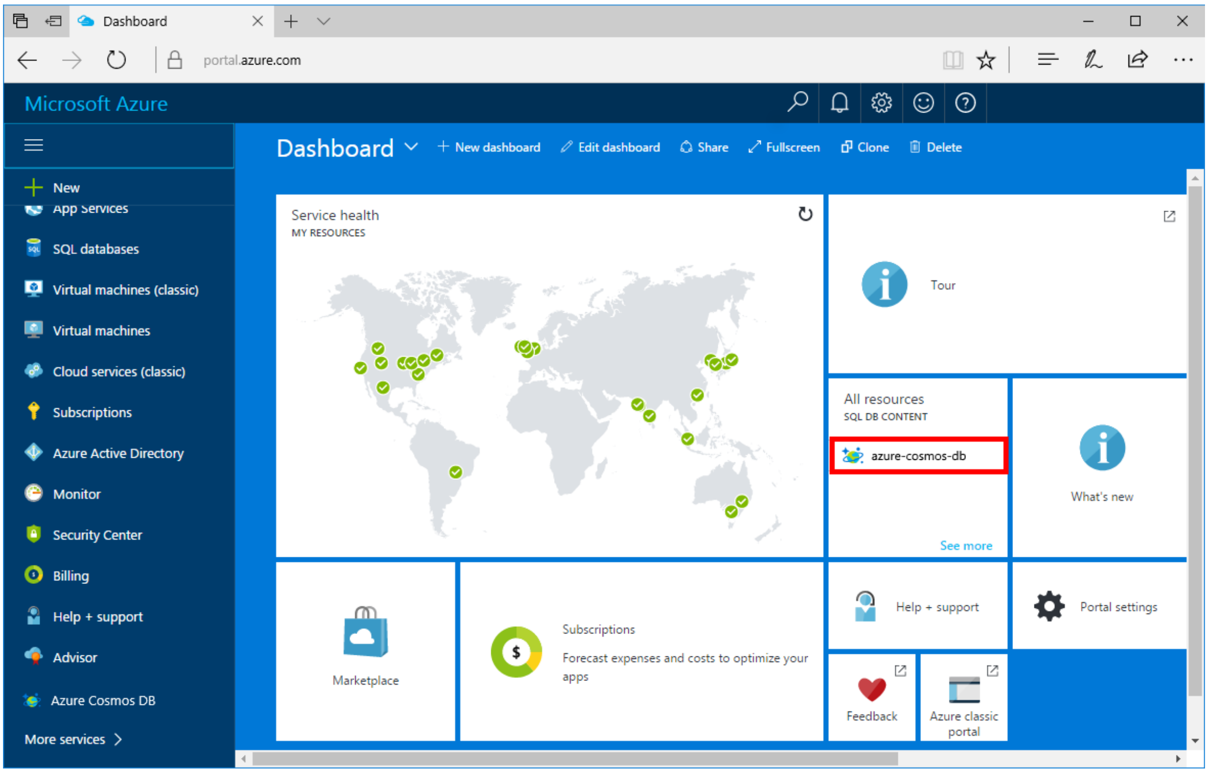
4. Click **Create** to create the account.
5. On the top toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

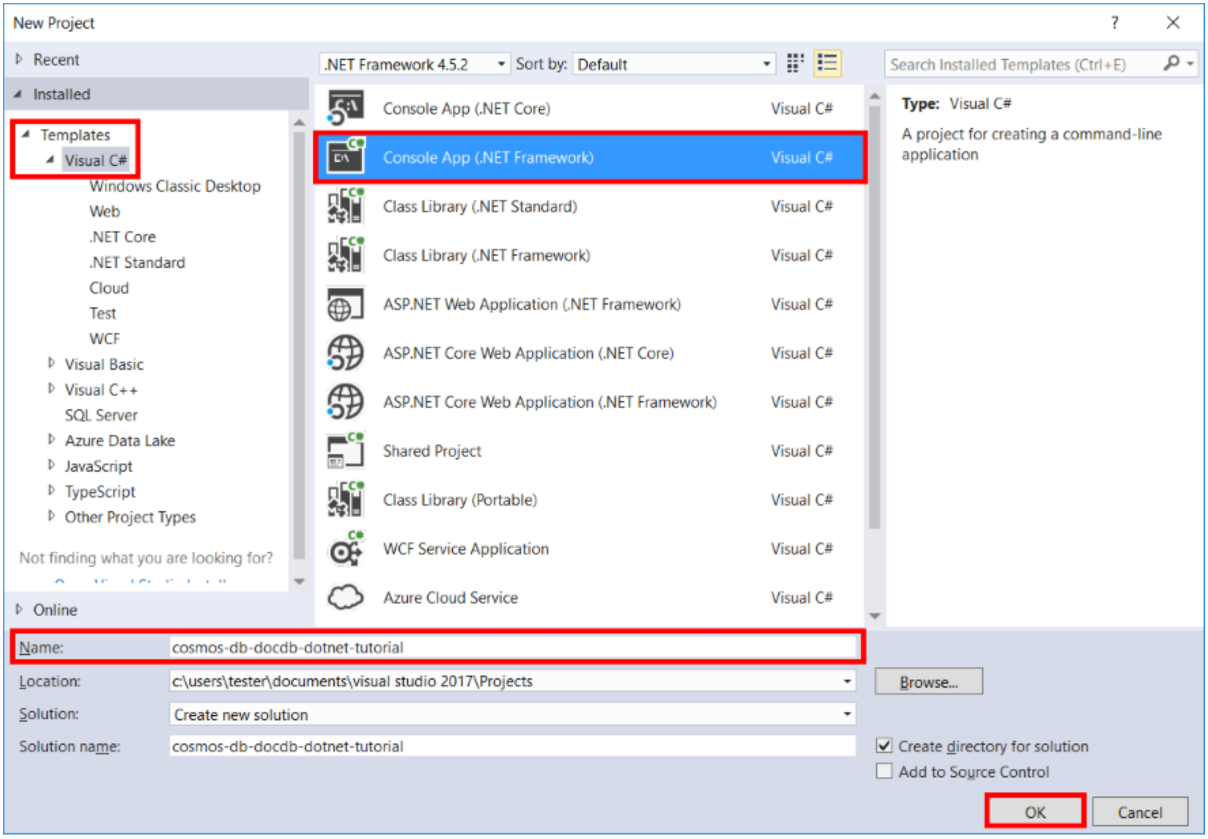
Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the **All Resources** tile.



Set up your Visual Studio solution

1. Open **Visual Studio** 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 App (.NET Framework)**, name your project, and then click **OK**.


New Project
Recent
Installed
Templates
Visual C#
Windows Classic Desktop
Web
.NET Core
.NET Standard
Cloud
Test
WCF
Visual Basic
Visual C++
SQL Server
Azure Data Lake
JavaScript
TypeScript
Other Project Types
Not finding what you are looking for?
Online
.NET Framework 4.5.2
Sort by: Default
Search Installed Templates (Ctrl+E)
Console App (.NET Core) Visual C#
Console App (.NET Framework) Visual C#
Class Library (.NET Standard) Visual C#
Class Library (.NET Framework) Visual C#
ASP.NET Web Application (.NET Framework) Visual C#
ASP.NET Core Web Application (.NET Core) Visual C#
ASP.NET Core Web Application (.NET Framework) Visual C#
Shared Project Visual C#
Class Library (Portable) Visual C#
WCF Service Application Visual C#
Azure Cloud Service Visual C#
Browse...
OK
Cancel

Type: Visual C#

A project for creating a command-line application

Name: cosmos-db-docdb-dotnet-tutorial

Location: c:\users\tester\documents\visual studio 2017\Projects

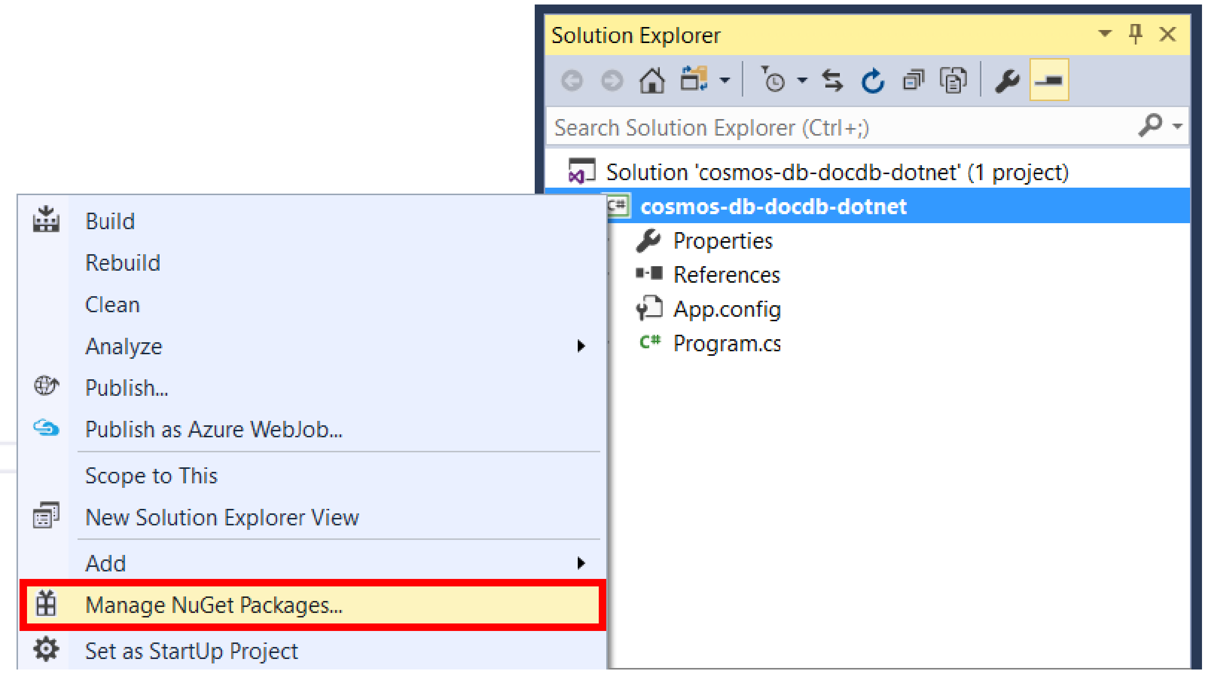
Solution: Create new solution

Solution name: cosmos-db-docdb-dotnet-tutorial

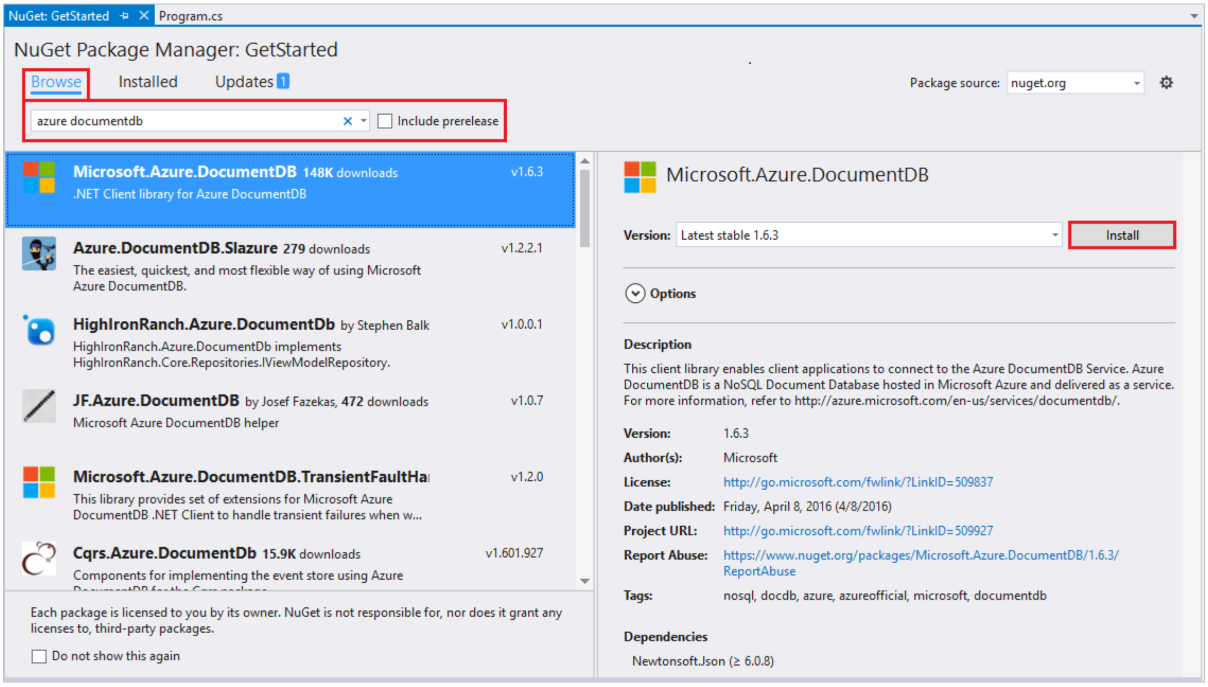
☑ Create directory for solution

☐ Add to Source Control

4. In the **Solution Explorer**, right click on your new console application, which is under your Visual Studio solution, and then click **Manage NuGet Packages...**



5. In the **NuGet** tab, click **Browse**, and type **documentdb** in the search box.
6. Within the results, find **Microsoft.Azure.DocumentDB** and click **Install**. The package ID for the Azure Cosmos DB Client Library is Microsoft.Azure.DocumentDB.



Microsoft.Azure.DocumentDB 148K downloads v1.6.3
.NET Client library for Azure DocumentDB

Azure.DocumentDB.Slazure 279 downloads v1.2.2.1
The easiest, quickest, and most flexible way of using Microsoft Azure DocumentDB.

HighIronRanch.Azure.DocumentDb by Stephen Balk v1.0.0.1
HighIronRanch.Azure.DocumentDb implements HighIronRanch.Core.Repositories.IViewModelRepository.

JF.Azure.DocumentDB by Josef Fazekas, 472 downloads v1.0.7
Microsoft Azure DocumentDB helper

Microsoft.Azure.DocumentDB.TransientFaultHa v1.2.0
This library provides set of extensions for Microsoft Azure DocumentDB .NET Client to handle transient failures when w...

Cqrs.Azure.DocumentDb 15.9K downloads v1.601.927
Components for implementing the event store using Azure

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

□ Do not show this again

Microsoft.Azure.DocumentDB

Version: Latest stable 1.6.3

Description

This client library enables client applications to connect to the Azure DocumentDB Service. Azure DocumentDB is a NoSQL Document Database hosted in Microsoft Azure and delivered as a service. For more information, refer to http://azure.microsoft.com/en-us/services/documentdb/.

Version: 1.6.3
Author(s): Microsoft
License: http://go.microsoft.com/fwlink/?LinkID=509837
Date published: Friday, April 8, 2016 (4/8/2016)
Project URL: http://go.microsoft.com/fwlink/?LinkID=509927
Report Abuse: https://www.nuget.org/packages/Microsoft.Azure.DocumentDB/1.6.3/ReportAbuse
Tags: nosql, docdb, azure, azureofficial, microsoft, documentdb

Dependencies
Newtonsoft.Json (≥ 6.0.8)

If you get a message about reviewing changes to the solution, click **OK**. If you get a message about license acceptance, click **I accept**.

Add references to your project

The remaining steps in this tutorial provide the DocumentDB API code snippets required to create and update Azure Cosmos DB resources in your project.

First, add these references to your application.

```
using System.Net;
using Microsoft.Azure.Documents;
using Microsoft.Azure.Documents.Client;
using Newtonsoft.Json;
```

Connect your app

Next, add these two constants and your *client* variable in your application.

```
private const string EndpointUrl = "<your endpoint URL>";
private const string PrimaryKey = "<your primary key>";
private DocumentClient client;
```

Then, head back to the Azure portal to retrieve your endpoint URL and primary key. The endpoint URL and primary key are necessary for your application to understand where to connect to, and for Azure Cosmos DB to trust your application's connection.

In the Azure portal, navigate to your Azure Cosmos DB account, click **Keys**, and then click **Read-write Keys**.

Copy the URI from the portal and paste it over `<your endpoint URL>` in the program.cs file. Then copy the PRIMARY KEY from the portal and paste it over `<your primary key>`. Be sure to remove the `<` and `>` from your values.

![Screen shot of the Azure portal used by the NoSQL tutorial to create a C# console application. Shows an Azure Cosmos DB account, with the KEYS highlighted on the Azure Cosmos DB account blade, and the URI and PRIMARY KEY values highlighted on the Keys blade][keys]

Instantiate the DocumentClient

Now, create a new instance of the **DocumentClient**.

```
DocumentClient client = new DocumentClient(new Uri(endpoint), authKey);
```

Create a database

Next, create an Azure Cosmos DB database by using the CreateDatabaseAsync method or CreateDatabaseIfNotExistsAsync method of the **DocumentClient** class from the DocumentDB .NET SDK. A database is the logical container of JSON document storage partitioned across collections.

```
await client.CreateDatabaseAsync(new Database { Id = "db" });
```

Decide on a partition key

Collections are containers for storing documents. They are logical resources and can span one or more physical partitions. A partition key is a property (or path) within your documents that is used to distribute your data among the servers or partitions. All documents with the same partition key are stored in the same partition.

Determining a partition key is an important decision to make before you create a collection. Partition keys are a property (or path) within your documents that can be used by Azure Cosmos DB to distribute your data among multiple servers or partitions. Cosmos DB hashes the partition key value and uses the hashed result to determine the partition in which to store the document. All documents with the same partition key are stored in the same partition, and partition keys cannot be changed once a collection is created.

For this tutorial, we're going to set the partition key to `/deviceId` so that the all the data for a single device is stored in a single partition. You want to choose a partition key that has a large number of values, each of which are used at about the same frequency to ensure Cosmos DB can load balance as your data grows and achieve the full throughput of the collection.

For more information about partitioning, see How to partition and scale in Azure Cosmos DB?

Create a collection

Now that we know our partition key, `/deviceId`, lets create a collection by using the CreateDocumentCollectionAsync method or CreateDocumentCollectionIfNotExistsAsync method of the **DocumentClient** class. A collection is a container of JSON documents and any associated JavaScript application logic.

> WARNING
>
> Creating a collection has pricing implications, as you are reserving throughput for the application to communicate with Azure Cosmos DB. For more details, please visit our pricing page

```
// Collection for device telemetry. Here the JSON property deviceId is used
// as the partition key to spread across partitions. Configured for 2500 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 = 2500 });
```

This method makes a REST API call to Azure Cosmos DB, and the service provisions a number of partitions based on the requested throughput. You can change the throughput of a collection as your performance needs evolve using the SDK or the Azure portal.

Create JSON documents

Let's insert some JSON documents into Azure Cosmos DB. A document can be created by using the CreateDocumentAsync method of the **DocumentClient** class. Documents are user-defined (arbitrary) JSON content. This sample class contains 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
    });
```

Read data

Let's read the document by its partition key and Id using the ReadDocumentAsync method. 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 data

Now let's update some data using the ReplaceDocumentAsync method.

```
// 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 data

Now lets delete a document by partition key and id by using the DeleteDocumentAsync method.

```
// Delete a document. The partition key is required.
await client.DeleteDocumentAsync(
  UriFactory.CreateDocumentUri("db", "coll", "XMS-001-FE24C"),
  new RequestOptions { PartitionKey = new PartitionKey("XMS-0001") });
```

Query partitioned collections

When you query data in partitioned collections, Azure Cosmos DB 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 Azure Cosmos DB 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.

Execute stored procedures

Lastly, you can 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 by adding the following code to your project.

```
await client.ExecuteStoredProcedureAsync<DeviceReading>(
   UriFactory.CreateStoredProcedureUri("db", "coll", "SetLatestStateAcrossReadings"),
   new RequestOptions { PartitionKey = new PartitionKey("XMS-001") },
   "XMS-001-FE24C");
```

And that's it! those are the main components of an Azure Cosmos DB application that uses a partition key to efficiently scale data distribution across partitions.

Clean up resources

If you're not going to continue to use this app, delete all resources created by this tutorial in the Azure portal with the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the unique name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this tutorial, you've done the following:

- Created an Azure Cosmos DB account
- Created a database and collection with a partition key
- Created JSON documents
- Updated a document
- Queried partitioned collections
- Ran a stored procedure
- Deleted a document
- Deleted a database

You can now proceed to the next tutorial and import additional data to your Cosmos DB account.

Import data into Azure Cosmos DB

Azure Cosmos DB: Connect to a MongoDB app using .NET

6/1/2017 • 5 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This tutorial demonstrates how to create an Azure Cosmos DB account using the Azure portal, and how to create a database and collection to store data using the MongoDB API.

This tutorial covers the following tasks:

- Create an Azure Cosmos DB account
- Update your connection string
- Create a MongoDB app on a virtual machine

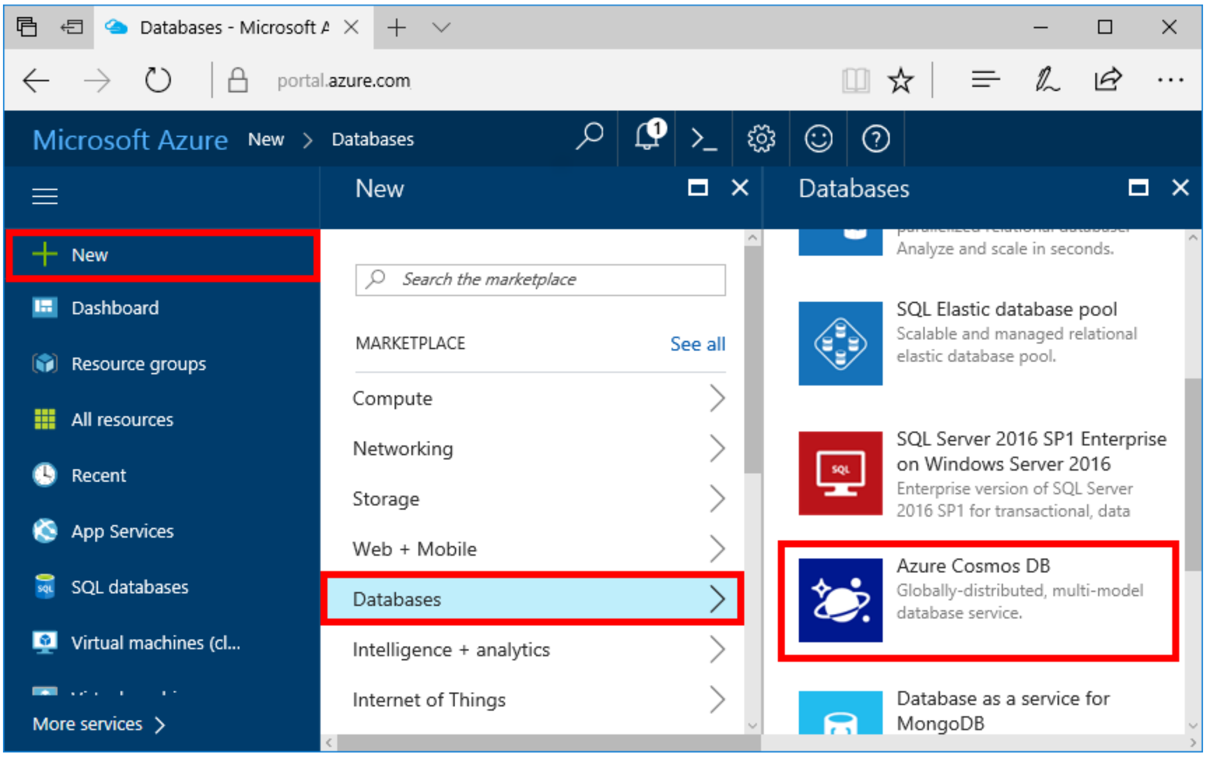
Create a database account

Let's start by creating an Azure Cosmos DB account in the Azure portal.

TIP

- Already have an Azure Cosmos DB account? If so, skip ahead to Set up your Visual Studio solution
- Did you have an Azure DocumentDB account? If so, your account is now an Azure Cosmos DB account and you can skip ahead to Set up your Visual Studio solution.
- If you are using the Azure Cosmos DB Emulator, please follow the steps at Azure Cosmos DB Emulator to setup the emulator and skip ahead to Set up your Visual Studio Solution.

1. In a new window, sign in to the Azure portal.
2. In the left menu, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

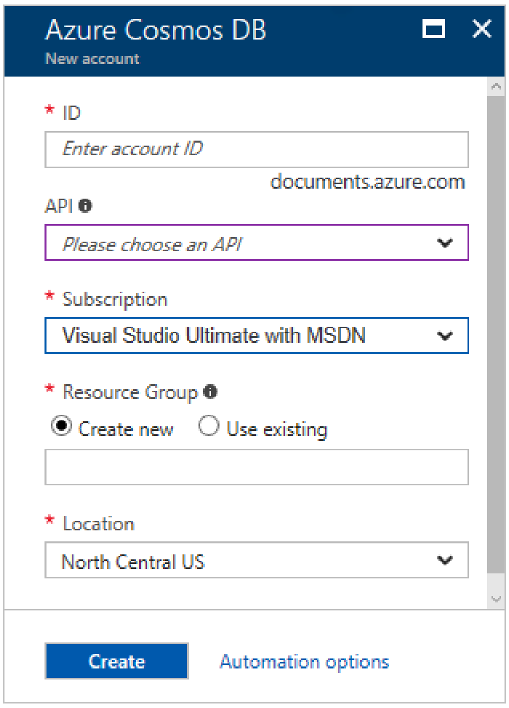

Databases - Microsoft A
portal.azure.com
Microsoft Azure New > Databases
New
Dashboard
Resource groups
All resources
Recent
App Services
SQL databases
Virtual machines (cl...
More services >
New
Search the marketplace
MARKETPLACE
See all
Compute
Networking
Storage
Web + Mobile
Databases
Intelligence + analytics
Internet of Things
Databases
Analyze and scale in seconds.
SQL Elastic database pool
Scalable and managed relational elastic database pool.
SQL Server 2016 SP1 Enterprise on Windows Server 2016
Enterprise version of SQL Server 2016 SP1 for transactional, data
Azure Cosmos DB
Globally-distributed, multi-model database service.
Database as a service for MongoDB

3. In the **New account** blade, specify the desired configuration for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick start we'll be programming against the MongoDB API so you'll choose **MongoDB** as you fill out the form. But if you have graph data for a social media app, document data from a catalog app, or key/value (table) data, realize that Azure Cosmos DB can provide a highly available, globally-distributed database service platform for all your mission-critical applications.

   Fill out the New account blade using the information in the screenshot as a guide . You will choose unique values as you set up your account so your values will not match the screenshot exactly


Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

* Subscription
Visual Studio Ultimate with MSDN

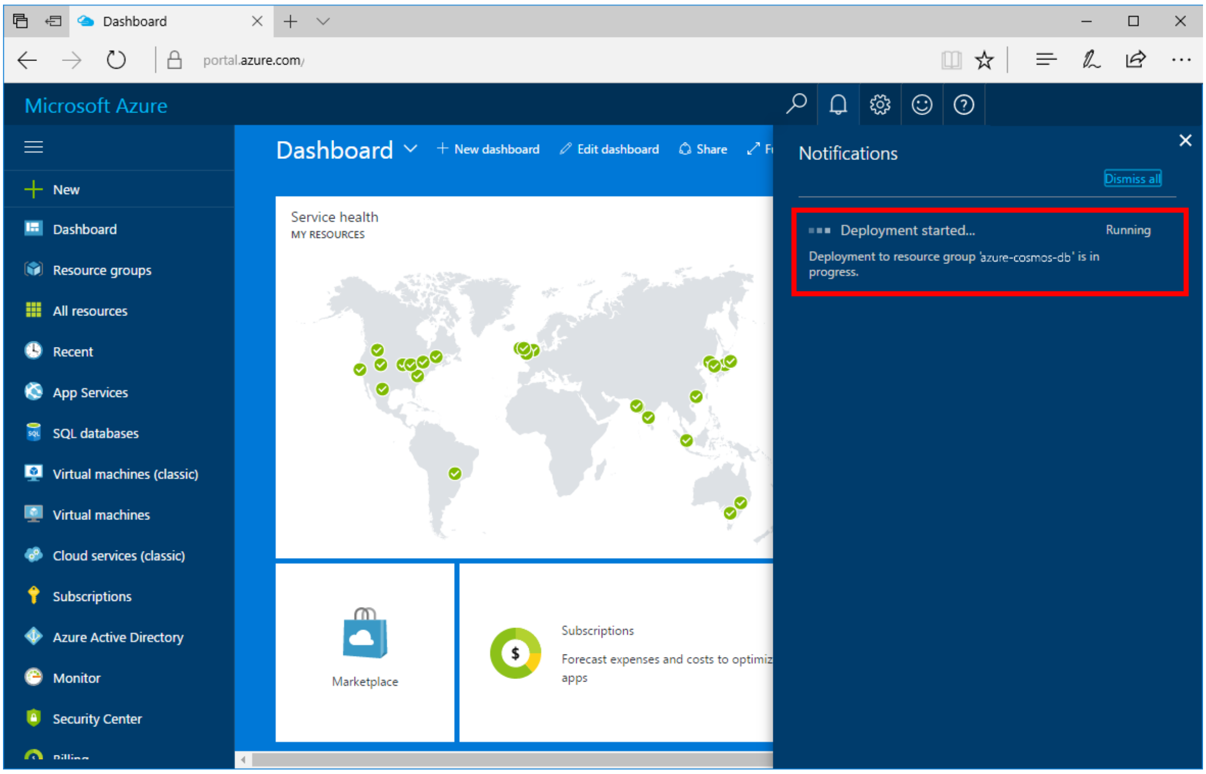
* Resource Group
● Create new ○ Use existing

* Location
North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name you choose to identify the Azure Cosmos DB account. *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID may contain only lowercase letters, numbers, and the '-' character, and must be between 3 and 50 characters.
API	MongoDB	We'll be programming against the MongoDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for the Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location closest to your users to give them the fastest access to the data.

4. Click **Create** to create the account.

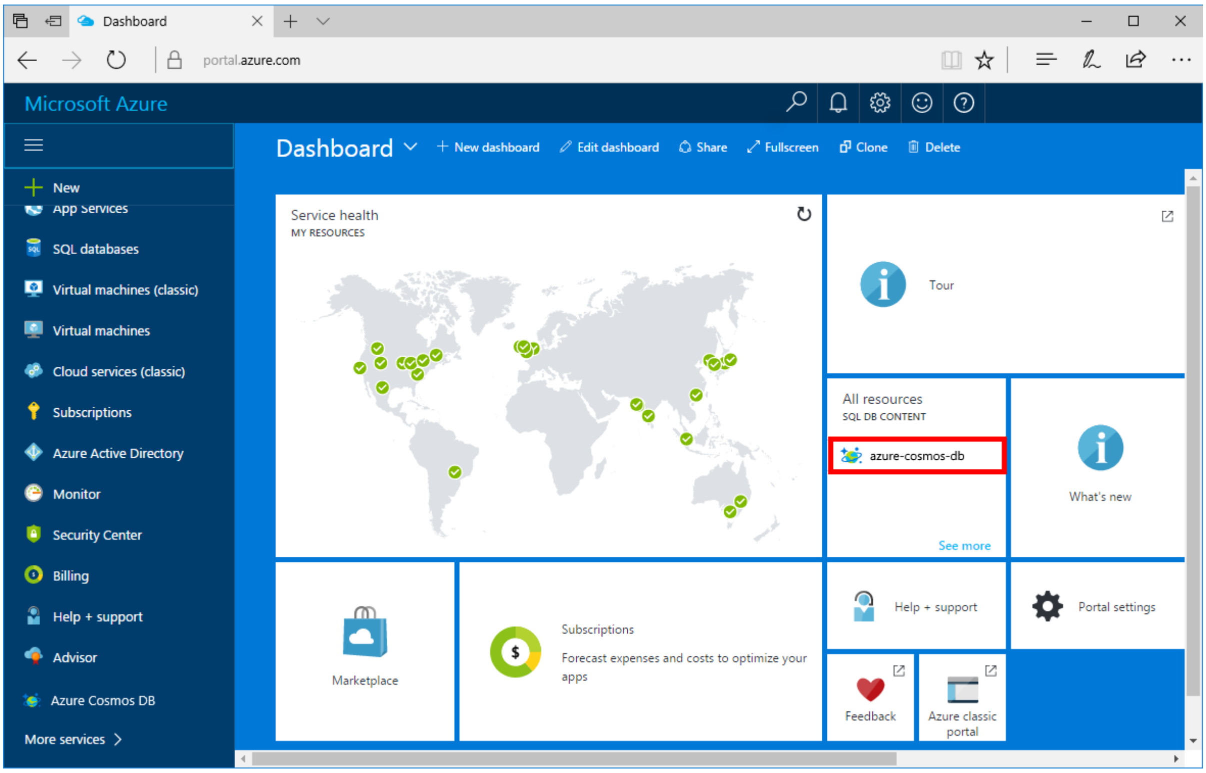
5. On the toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the All Resources tile.



Update your connection string

1. In the Azure portal, in the **Azure Cosmos DB** page, select the API for MongoDB account.
2. In the left bar of the account blade, click **Quick start**.
3. Choose your platform (*.NET driver*, *Node.js driver*, *MongoDB Shell*, *Java driver*, *Python driver*). If you don't see your driver or tool listed, don't worry, we continuously document more connection code snippets.
4. Copy and paste the code snippet into your MongoDB app, and you are ready to go.

Set up your MongoDB app

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 an API for MongoDB account.

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 from the Keys page in the Azure portal:

```
private string userName = "<your user name>";
private string host = "<your host>";
private string password = "<your password>";
```

3. Use the app!

Clean up resources

If you're not going to continue to use this app, use the following steps to delete all resources created by this tutorial in the Azure portal.

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this tutorial, you've done the following:

- Create an Azure Cosmos DB account
- Update your connection string
- Create a MongoDB app on a virtual machine

You can proceed to the next tutorial and import your MongoDB data to Azure Cosmos DB.

Import MongoDB data into Azure Cosmos DB

Azure Cosmos DB: Develop with the Table API in .NET

6/1/2017 • 15 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This tutorial covers the following tasks:

- Create an Azure Cosmos DB account
- Enable functionality in the app.config file
- Create a table using the Table API (preview)
- Add an entity to a table
- Insert a batch of entities
- Retrieve a single entity
- Query entities using automatic secondary indexes
- Replace an entity
- Delete an entity
- Delete a table

Tables in Azure Cosmos DB

Azure Cosmos DB provides the Table API (preview) for applications that need a key-value store with a schema-less design. Azure Table storage SDKs and REST APIs can be used to work with Azure Cosmos DB. You can use Azure Cosmos DB to create tables with high throughput requirements. Azure Cosmos DB supports throughput-optimized tables (informally called "premium tables"), currently in public preview.

You can continue to use Azure Table storage for tables with high storage and lower throughput requirements. Azure Cosmos DB will introduce support for storage-optimized tables in a future update, and existing and new Azure Table storage accounts will be seamlessly upgraded to Azure Cosmos DB.

If you currently use Azure Table storage, you gain the following benefits with the "premium table" preview:

- Turn-key global distribution with multi-homing and automatic and manual failovers
- Support for automatic schema-agnostic indexing against all properties ("secondary indexes"), and fast queries
- Support for independent scaling of storage and throughput, across any number of regions
- Support for dedicated throughput per table that can be scaled from hundreds to millions of requests per second
- Support for five tunable consistency levels to trade off availability, latency, and consistency based on your application needs
- 99.99% availability within a single region, and ability to add more regions for higher availability, and industry-leading comprehensive SLAs on general availability
- Work with the existing Azure storage .NET SDK, and no code changes to your application

During the preview, Azure Cosmos DB supports the Table API using the .NET SDK. You can download the Azure Storage Preview SDK from NuGet, that has the same classes and method signatures as the Azure Storage SDK, but also can connect to Azure Cosmos DB accounts using the Table API.

To learn more about complex Azure Table storage tasks, see:

- Introduction to Azure Cosmos DB: Table API
- The Table service reference documentation for complete details about available APIs Storage Client Library for .NET reference

About this tutorial

This tutorial is for developers who are familiar with the Azure Table storage SDK, and would like to use the premium features available using Azure Cosmos DB. It is based on Get Started with Azure Table storage using .NET and shows how to take advantage of additional capabilities like secondary indexes, provisioned throughput, and multi-homing. We cover how to use the Azure portal to create an Azure Cosmos DB account, and then build and deploy a Table application. We also walk through .NET examples for creating and deleting a table, and inserting, updating, deleting, and querying table data.

If you don't already have Visual Studio 2017 installed, you can download and use the **free** Visual Studio 2017 Community Edition. Make sure that you enable **Azure development** during the Visual Studio setup.

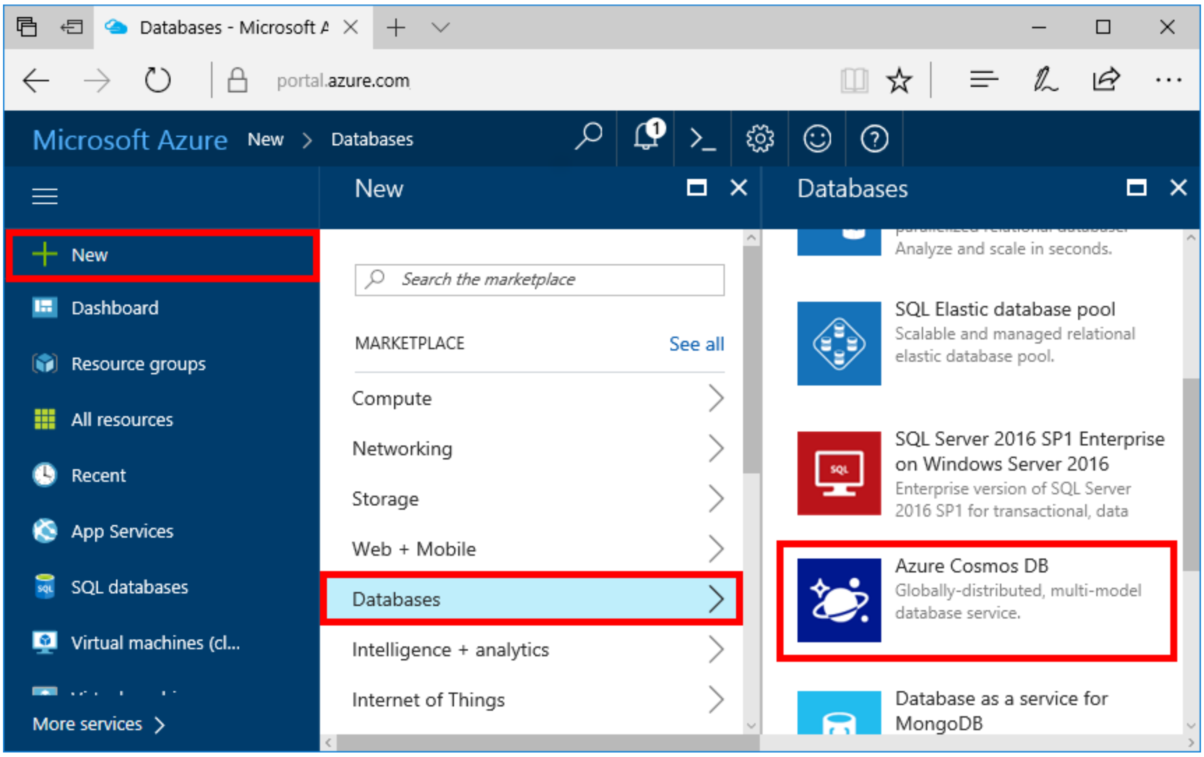
If you don't have an Azure subscription, create a free account before you begin.

Create a database account

Let's start by creating an Azure Cosmos DB account in the Azure portal.

> TIP
>
> - Already have an Azure Cosmos DB account? If so, skip ahead to Set up your Visual Studio solution.
> - Did you have an Azure DocumentDB account? If so, your account is now an Azure Cosmos DB account and you can skip ahead to Set up your Visual Studio solution.
> - If you are using the Azure Cosmos DB Emulator, please follow the steps at Azure Cosmos DB Emulator to setup the emulator and skip ahead to Set up your Visual Studio Solution.

1. In a new window, sign in to the Azure portal.
2. In the left menu, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

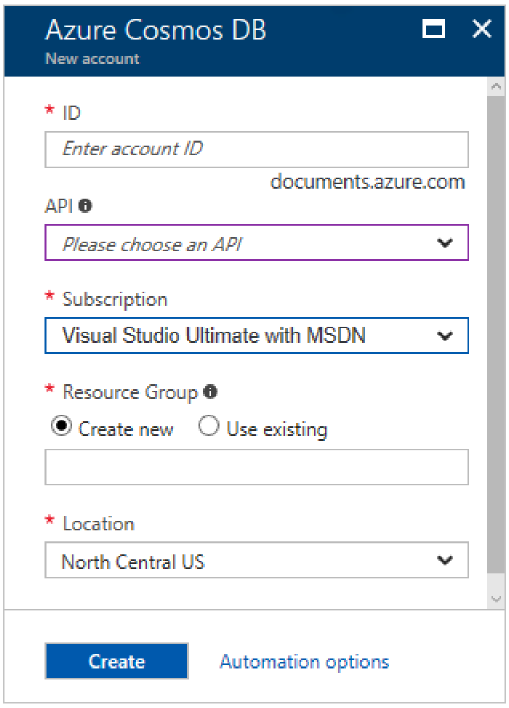


3. In the **New account** blade, specify the desired configuration for the Azure Cosmos DB account.

With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

In this quick start we'll be programming against the Table API so you'll choose **Table (key-value)** as you fill out the form. But if you have graph data for a social media app, document data from a catalog app, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally-distributed database service platform for all your mission-critical applications.

Fill out the New account blade using the information in the screenshot as a guide. You will choose unique values as you set up your account so your values will not match the screenshot exactly.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

* Subscription
Visual Studio Ultimate with MSDN

* Resource Group
● Create new ○ Use existing

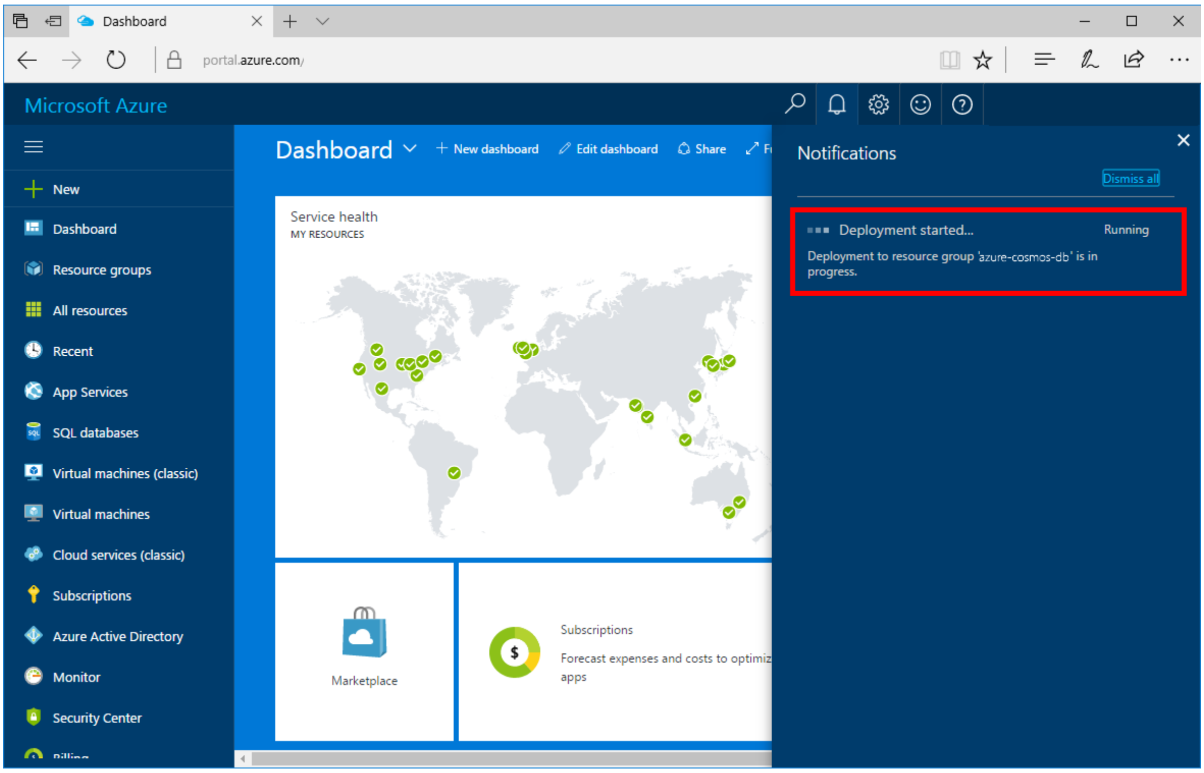
* Location
North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name you choose to identify the Azure Cosmos DB account. *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID may contain only lowercase letters, numbers, and the '-' character, and must be between 3 and 50 characters.
API	Table (key-value)	We'll be programming against the Table API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for the Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource group name for your account. For simplicity, you can use the same name as your ID.

SETTING	SUGGESTED VALUE	DESCRIPTION
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location closest to your users to give them the fastest access to the data.

4. Click **Create** to create the account.

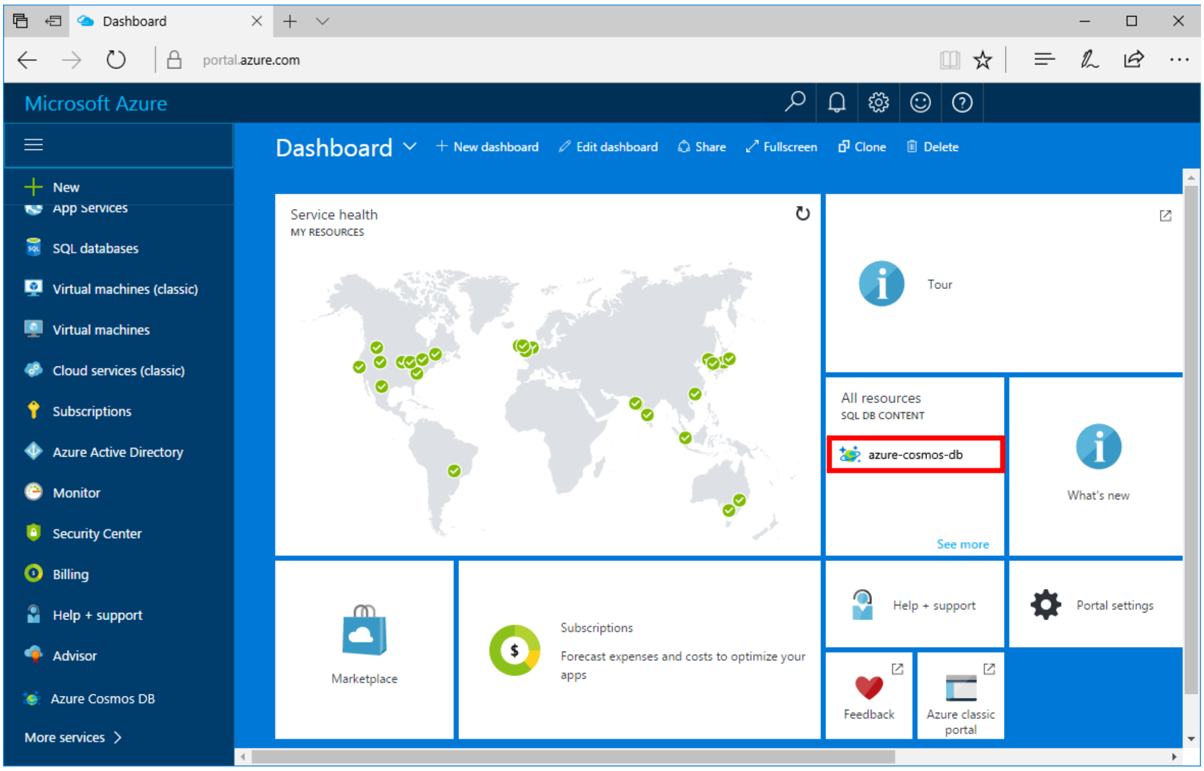
5. On the toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the All Resources tile.



Clone the sample application

Now let's clone a Table app from github, set the connection string, and run it.

1. Open a git terminal window, such as git bash, and `cd` to a working directory.

2. Run the following command to clone the sample repository.

   ```
   git clone https://github.com/Azure-Samples/azure-cosmos-db-table-dotnet-getting-started
   ```

3. Then open the solution file in Visual Studio.

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. In the Azure portal, in your Azure Cosmos DB account, in the left navigation click **Keys**, and then click **Read-write Keys**. You'll use the copy buttons on the right side of the screen to copy the connection string into the app.config file in the next step.

2. In Visual Studio, open the app.config file.

3. Copy your URI value from the portal (using the copy button) and make it the value of the account-key in app.config. Use the account name created earlier for account-name in app.config.

```
<add key="StorageConnectionString" value="DefaultEndpointsProtocol=https;AccountName=account-name;AccountKey=account-key;TableEndpoint=https://account-name.documents.azure.com" />
```

NOTE

To use this app with standard Azure Table Storage, you need to change the connection string in `app.config file`. Use the account name as Table-account name and key as Azure Storage Primary key.

```
<add key="StorageConnectionString" value="DefaultEndpointsProtocol=https;AccountName=account-name;AccountKey=account-key;EndpointSuffix=core.windows.net" />
```

Build and deploy the app

1. In Visual Studio, right-click on the project in **Solution Explorer** and then click **Manage NuGet Packages**.

2. In the NuGet **Browse** box, type ***WindowsAzure.Storage-PremiumTable***. Check **Include prerelease versions**.

3. From the results, install the **WindowsAzure.Storage-PremiumTable** and choose the preview build `0.0.1-preview`. This action installs the Azure Table storage package and all dependencies.

4. Click CTRL + F5 to run the application.

You can now go back to Data Explorer and see query, modify, and work with this table data.

NOTE

To use this app with an Azure Cosmos DB Emulator, you just need to change the connection string in `app.config file`. Use the below value for emulator.

```
<add key="StorageConnectionString" value=DefaultEndpointsProtocol=https;AccountName=localhost;AccountKey=<insertkey>==;TableEndpoint=https://localhost -->
```

Azure Cosmos DB capabilities

Azure Cosmos DB supports a number of capabilities that are not available in the Azure Table storage API. The new functionality can be enabled via the following `appSettings` configuration values. We did not introduce any new signatures or overloads to the preview Azure Storage SDK. This allows you to connect to both standard and premium tables, and work with other Azure Storage services like Blobs and Queues.

KEY	DESCRIPTION
TableConnectionMode	Azure Cosmos DB supports two connectivity modes. In `Gateway` mode, requests are always made to the Azure Cosmos DB gateway, which forwards it to the corresponding data partitions. In `Direct` connectivity mode, the client fetches the mapping of tables to partitions, and requests are made directly against data partitions. We recommend `Direct`, the default.
TableConnectionProtocol	Azure Cosmos DB supports two connection protocols - `Https` and `Tcp`. `Tcp` is the default, and recommended because it is more lightweight.
TablePreferredLocations	Comma-separated list of preferred (multi-homing) locations for reads. Each Azure Cosmos DB account can be associated with 1-30+ regions. Each client instance can specify a subset of these regions in the preferred order for low latency reads. The regions must be named using their display names, for example, `West US`. Also see Multi-homing APIs.
TableConsistencyLevel	You can trade off between latency, consistency, and availability by choosing between five well-defined consistency levels: `Strong`, `Session`, `Bounded-Staleness`, `ConsistentPrefix`, and `Eventual`. Default is `Session`. The choice of consistency level makes a significant performance difference in multi-region setups. See Consistency levels for details.
TableThroughput	Reserved throughput for the table expressed in request units (RU) per second. Single tables can support 100s-millions of RU/s. See Request units. Default is `400`
TableIndexingPolicy	Consistent and automatic secondary indexing of all columns within tables
TableQueryMaxItemCount	Configure the maximum number of items returned per table query in a single round trip. Default is `-1`, which lets Azure Cosmos DB dynamically determine the value at runtime.
TableQueryEnableScan	If the query cannot use the index for any filter, then run it anyway via a scan. Default is `false`.
TableQueryMaxDegreeOfParallelism	The degree of parallelism for execution of a cross-partition query. `0` is serial with no pre-fetching, `1` is serial with pre-fetching, and higher values increase the rate of parallelism. Default is `-1`, which lets Azure Cosmos DB dynamically determine the value at runtime.

To change the default value, open the `app.config` file from Solution Explorer in Visual Studio. Add the contents of the `<appSettings>` element shown below. Replace `account-name` with the name of your storage account, and `account-key` with your account access key.

```
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.2" />
  </startup>
  <appSettings>
   <!-- Client options -->
   <add key="StorageConnectionString" value="DefaultEndpointsProtocol=https;AccountName=account-name;AccountKey=account-key;
TableEndpoint=https://account-name.documents.azure.com" />
   <add key="TableConnectionMode" value="Direct"/>
   <add key="TableConnectionProtocol" value="Tcp"/>
   <add key="TablePreferredLocations" value="East US, West US, North Europe"/>
   <add key="TableConsistencyLevel" value="Eventual"/>

   <!--Table creation options -->
   <add key="TableThroughput" value="700"/>
   <add key="TableIndexingPolicy" value="{""indexingMode"": ""Consistent""}">

   <!-- Table query options -->
   <add key="TableQueryMaxItemCount" value="-1"/>
   <add key="TableQueryEnableScan" value="false"/>
   <add key="TableQueryMaxDegreeOfParallelism" value="-1"/>
   <add key="TableQueryContinuationTokenLimitInKb" value="16"/>

  </appSettings>
</configuration>
```

Let's make a quick review of what's happening in the app. Open the `Program.cs` file and you find that these lines of code create the Table resources.

Create the table client

You initialize a `CloudTableClient` to connect to the table account.

```
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();
```

This client is initialized using the `TableConnectionMode`, `TableConnectionProtocol`, `TableConsistencyLevel`, and `TablePreferredLocations` configuration values if specified in the app settings.

Create a table

Then, you create a table using `CloudTable`. Tables in Azure Cosmos DB can scale independently in terms of storage and throughput, and partitioning is handled automatically by the service. Azure Cosmos DB supports both fixed size and unlimited tables. See Partitioning in Azure Cosmos DB for details.

```
CloudTable table = tableClient.GetTableReference("people");

table.CreateIfNotExists();
```

There is an important difference in how tables are created. Azure Cosmos DB reserves throughput, unlike Azure storage's consumption-based model for transactions. The reservation model has two key benefits:

- Your throughput is dedicated/reserved, so you never get throttled if your request rate is at or below your provisioned throughput
- The reservation model is more cost effective for throughput-heavy workloads

You can configure the default throughput by configuring the setting for `TableThroughput` in terms of RU (request units) per second.

A read of a 1-KB entity is normalized as 1 RU, and other operations are normalized to a fixed RU value based on their CPU, memory, and IOPS consumption. Learn more about Request units in Azure Cosmos DB.

> NOTE
>
> While Table storage SDK does not currently support modifying throughput, you can change the throughput instantaneously at any time using the Azure portal or Azure CLI.

Next, we walk through the simple read and write (CRUD) operations using the Azure Table storage SDK. This tutorial demonstrates predictable low single-digit millisecond latencies and fast queries provided by Azure Cosmos DB.

Add an entity to a table

Entities in Azure Table storage extend from the `TableEntity` class and must have `PartitionKey` and `RowKey` properties. Here's a sample definition for a customer entity.

```
public class CustomerEntity : TableEntity
{
    public CustomerEntity(string lastName, string firstName)
    {
        this.PartitionKey = lastName;
        this.RowKey = firstName;
    }

    public CustomerEntity() { }

    public string Email { get; set; }

    public string PhoneNumber { get; set; }
}
```

The following snippet shows how to insert an entity with the Azure storage SDK. Azure Cosmos DB is designed for guaranteed low latency at any scale, across the world.

Writes complete <15 ms at p99 and ~6 ms at p50 for applications running in the same region as the Azure Cosmos DB account. And this duration accounts for the fact that writes are acknowledged back to the client only after they are synchronously replicated, durably committed, and all content is indexed.

The Table API for Azure Cosmos DB is in preview. At general availability, the p99 latency guarantees are backed by SLAs like other Azure Cosmos DB APIs.

```
// Create a new customer entity.
CustomerEntity customer1 = new CustomerEntity("Harp", "Walter");
customer1.Email = "Walter@contoso.com";
customer1.PhoneNumber = "425-555-0101";

// Create the TableOperation object that inserts the customer entity.
TableOperation insertOperation = TableOperation.Insert(customer1);

// Execute the insert operation.
table.Execute(insertOperation);
```

Insert a batch of entities

Azure Table storage supports a batch operation API, that lets you combine updates, deletes, and inserts in the same single batch operation. Azure Cosmos DB does not have some of the limitations on the batch API as Azure Table storage. For example, you can perform multiple reads within a batch, you can perform multiple writes to the same entity within a batch, and there is no limit on 100 operations per batch.

```
// Create the batch operation.
TableBatchOperation batchOperation = new TableBatchOperation();

// Create a customer entity and add it to the table.
CustomerEntity customer1 = new CustomerEntity("Smith", "Jeff");
customer1.Email = "Jeff@contoso.com";
customer1.PhoneNumber = "425-555-0104";

// Create another customer entity and add it to the table.
CustomerEntity customer2 = new CustomerEntity("Smith", "Ben");
customer2.Email = "Ben@contoso.com";
customer2.PhoneNumber = "425-555-0102";

// Add both customer entities to the batch insert operation.
batchOperation.Insert(customer1);
batchOperation.Insert(customer2);

// Execute the batch operation.
table.ExecuteBatch(batchOperation);
```

Retrieve a single entity

Retrieves (GETs) in Azure Cosmos DB complete <10 ms at p99 and ~1 ms at p50 in the same Azure region. You can add as many regions to your account for low latency reads, and deploy applications to read from their local region ("multi-homed") by setting `TablePreferredLocations`.

You can retrieve a single entity using the following snippet:

```
// Create a retrieve operation that takes a customer entity.
TableOperation retrieveOperation = TableOperation.Retrieve<CustomerEntity>("Smith", "Ben");

// Execute the retrieve operation.
TableResult retrievedResult = table.Execute(retrieveOperation);
```

TIP

Learn about multi-homing APIs at Developing with multiple regions

Query entities using automatic secondary indexes

Tables can be queried using the `TableQuery` class. Azure Cosmos DB has a write-optimized database engine that automatically indexes all columns within your table. Indexing in Azure Cosmos DB is agnostic to schema. Therefore, even if your schema is different between rows, or if the schema evolves over time, it is automatically indexed. Since Azure Cosmos DB supports automatic secondary indexes, queries against any property can use the index and be served efficiently.

```
CloudTable table = tableClient.GetTableReference("people");

// Filter against a property that's not partition key or row key
TableQuery<CustomerEntity> emailQuery = new TableQuery<CustomerEntity>().Where(
   TableQuery.GenerateFilterCondition("Email", QueryComparisons.Equal, "Ben@contoso.com"));

foreach (CustomerEntity entity in table.ExecuteQuery(emailQuery))
{
   Console.WriteLine("{0}, {1}\t{2}\t{3}", entity.PartitionKey, entity.RowKey,
      entity.Email, entity.PhoneNumber);
}
```

In preview, Azure Cosmos DB supports the same query functionality as Azure Table storage for the Table API. Azure Cosmos DB also supports sorting, aggregates, geospatial query, hierarchy, and a wide range of built-in functions. The additional functionality will be provided in the Table API in a future service update. See Azure Cosmos DB query for an overview of these capabilities.

Replace an entity

To update an entity, retrieve it from the Table service, modify the entity object, and then save the changes back to the Table service. The following code changes an existing customer's phone number.

```
TableOperation updateOperation = TableOperation.Replace(updateEntity);
table.Execute(updateOperation);
```

Similarly, you can perform `InsertOrMerge` or `Merge` operations.

Delete an entity

You can easily delete an entity after you have retrieved it by using the same pattern shown for updating an entity. The following code retrieves and deletes a customer entity.

```
TableOperation deleteOperation = TableOperation.Delete(deleteEntity);
table.Execute(deleteOperation);
```

Delete a table

Finally, the following code example deletes a table from a storage account. You can delete and recreate a table immediately with Azure Cosmos DB.

```
CloudTable table = tableClient.GetTableReference("people");
table.DeleteIfExists();
```

Clean up resources

If you're not going to continue to use this app, use the following steps to delete all resources created by this tutorial in the Azure portal.

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this tutorial, we covered how to get started using Azure Cosmos DB with the Table API, and you've done the following:

- Created an Azure Cosmos DB account
- Enabled functionality in the app.config file
- Created a table
- Added an entity to a table
- Inserted a batch of entities
- Retrieved a single entity
- Queried entities using automatic secondary indexes
- Replaced an entity
- Deleted an entity
- Deleted a table

You can now proceed to the next tutorial and learn more about querying table data.

Query with the Table API

Azure Cosmos DB: Develop with the Graph API in .NET

6/1/2017 • 10 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This tutorial demonstrates how to create an Azure Cosmos DB account using the Azure portal and how to create a graph database and container. The application then creates a simple social network with four people using the Graph API (preview), then traverses and queries the graph using Gremlin.

This tutorial covers the following tasks:

- Create an Azure Cosmos DB account
- Create a graph database and container
- Serialize vertices and edges to .NET objects
- Add vertices and edges
- Query the graph using Gremlin

Graphs in Azure Cosmos DB

You can use Azure Cosmos DB to create, update, and query graphs using the Microsoft.Azure.Graphs library. The Microsoft.Azure.Graph library provides a single extension method `CreateGremlinQuery<T>` on top of the `DocumentClient` class to execute Gremlin queries.

Gremlin is a functional programming language that supports write operations (DML) and query and traversal operations. We cover a few examples in this article to get your started with Gremlin. See Gremlin queries for a detailed walkthrough of Gremlin capabilities available in Azure Cosmos DB.

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.

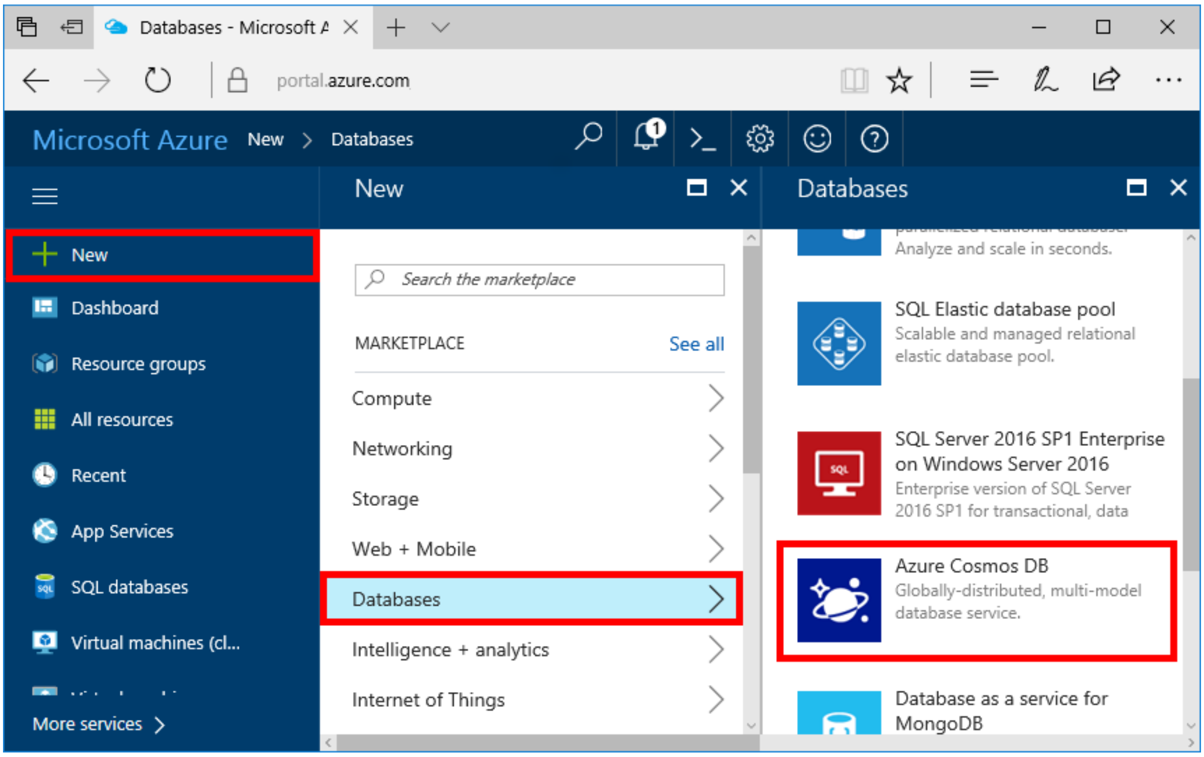
Create database account

Let's start by creating an Azure Cosmos DB account in the Azure portal.

TIP

- Already have an Azure Cosmos DB account? If so, skip ahead to Set up your Visual Studio solution
- Did you have an Azure DocumentDB account? If so, your account is now an Azure Cosmos DB account and you can skip ahead to Set up your Visual Studio solution.
- If you are using the Azure Cosmos DB Emulator, please follow the steps at Azure Cosmos DB Emulator to setup the emulator and skip ahead to Set up your Visual Studio Solution.

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

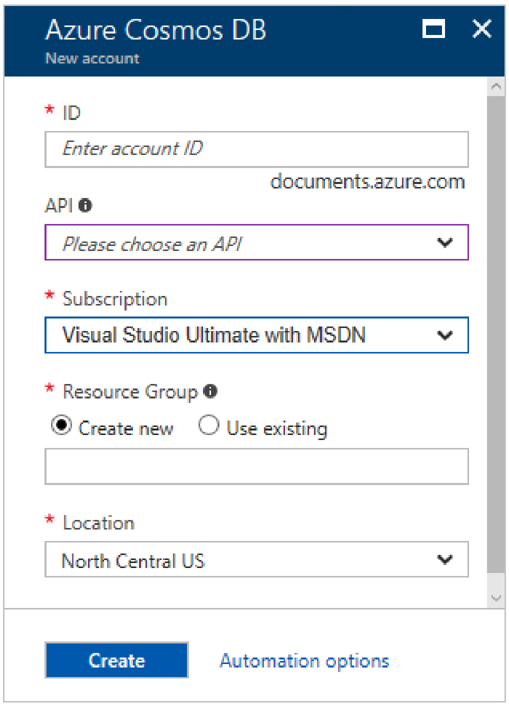


3. In the **New account** blade, specify the desired configuration for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article, we program against the Graph API, so choose **Gremlin (graph)** as you fill out the form. If you have document data from a catalog app, key/value (table) data, or data that's migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

   On the **New account** blade, complete the fields with the information in the following screenshot as a guide only. Because your own values will not match those in the screenshot, be sure to choose unique values as you set up your account.


Azure Cosmos DB
New account

* ID: Enter account ID .documents.azure.com

API: Please choose an API

* Subscription: Visual Studio Ultimate with MSDN

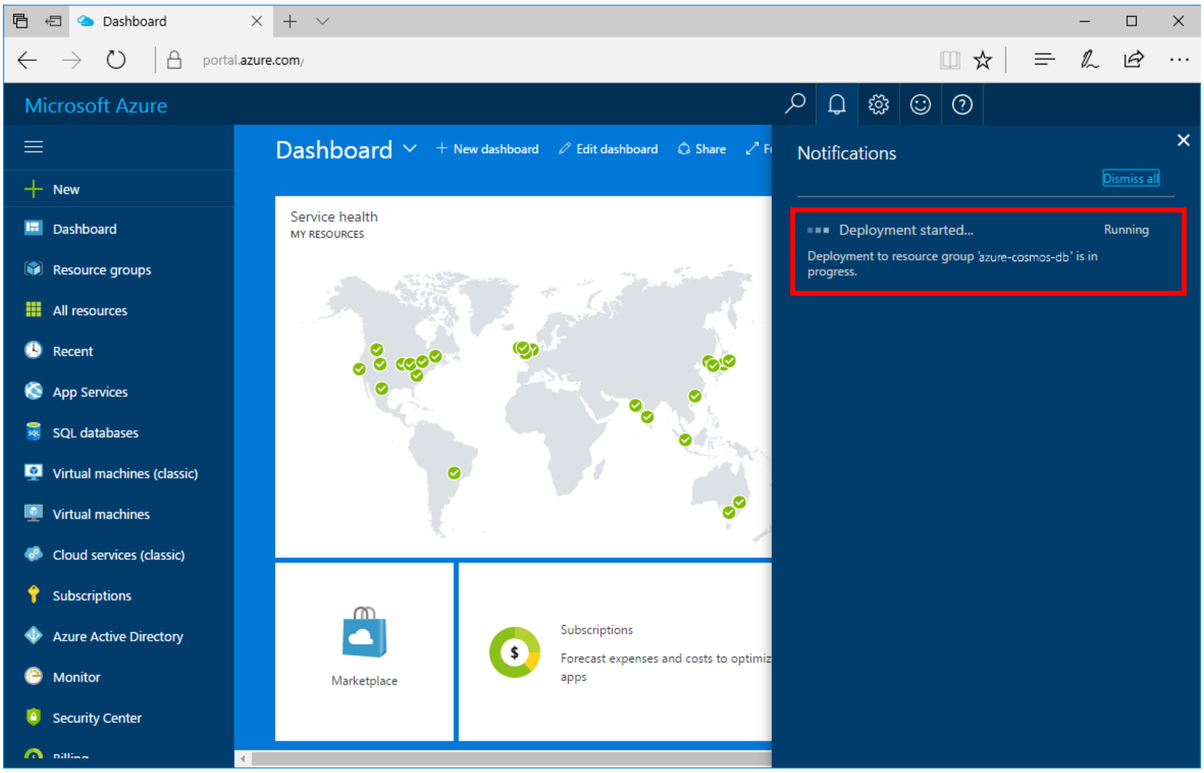
* Resource Group: ● Create new ○ Use existing

* Location: North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that you choose to identify the Azure Cosmos DB account. Because *documents.azure.com* is appended to the ID that you provide to create your URI, use a unique but identifiable ID. The ID must contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 to 50 characters.
API	Gremlin (graph)	We program against the Graph API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for the Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location closest to your users to give them the fastest access to the data.

4. Click **Create** to create the account.

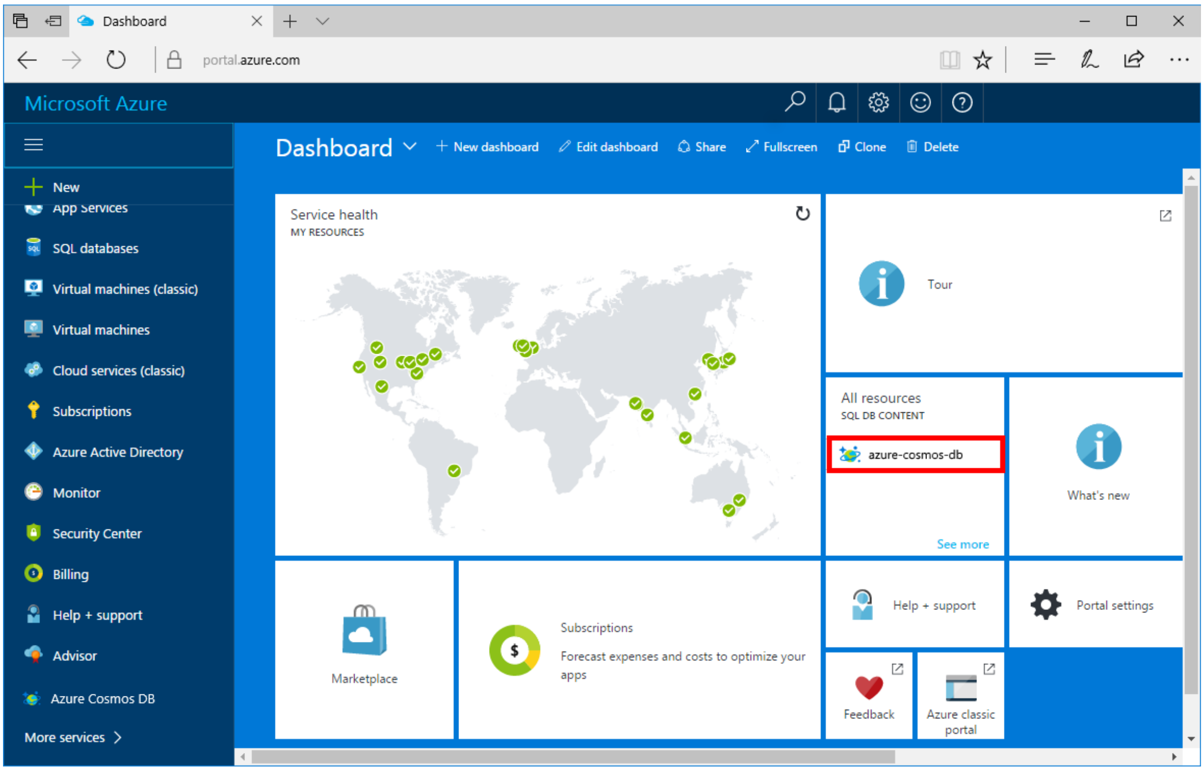
5. On the toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the **All Resources** tile.



Set up your Visual Studio solution

1. Open **Visual Studio** 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 App (.NET Framework)**, 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, and then click **Manage NuGet Packages...**
5. In the **NuGet** tab, click **Browse**, and type **Microsoft.Azure.Graphs** in the search box, and check the **Include prerelease versions**.
6. Within the results, find **Microsoft.Azure.Graphs** and click **Install**.

   If you get a message about reviewing changes to the solution, click **OK**. If you get a message about license acceptance, click **I accept**.

   The `Microsoft.Azure.Graphs` library provides a single extension method `CreateGremlinQuery<T>` for executing Gremlin operations. Gremlin is a functional programming language that supports write operations (DML) and query and traversal operations. We cover a few examples in this article to get your started with Gremlin. Gremlin queries has a detailed walkthrough of Gremlin capabilities in Azure Cosmos DB.

Connect your app

Add these two constants and your *client* variable in your application.

```
string endpoint = ConfigurationManager.AppSettings["Endpoint"];
string authKey = ConfigurationManager.AppSettings["AuthKey"];
```

Next, head back to the Azure portal to retrieve your endpoint URL and primary key. The endpoint URL and primary key are necessary for your application to understand where to connect to, and for Azure Cosmos DB to trust your application's connection.

In the Azure portal, navigate to your Azure Cosmos DB account, click **Keys**, and then click **Read-write Keys**.

Copy the URI from the portal and paste it over `Endpoint` in the endpoint property above. Then copy the PRIMARY KEY from the portal and paste it into the `AuthKey` property above.

![Screen shot of the Azure portal used by the tutorial to create a C# application. Shows an Azure Cosmos DB account the KEYS button highlighted on the Azure Cosmos DB navigation , and the URI and PRIMARY KEY values highlighted on the Keys blade][keys]

Instantiate the DocumentClient

Next, create a new instance of the **DocumentClient**.

```
DocumentClient client = new DocumentClient(new Uri(endpoint), authKey);
```

Create a database

Now, create an Azure Cosmos DB database by using the CreateDatabaseAsync method or CreateDatabaseIfNotExistsAsync method of the **DocumentClient** class from the DocumentDB .NET SDK.

```
Database database = await client.CreateDatabaseIfNotExistsAsync(new Database { Id = "graphdb" });
```

Create a graph

Next, create a graph container by using the using the CreateDocumentCollectionAsync method or CreateDocumentCollectionIfNotExistsAsync method of the **DocumentClient** class. A collection is a container of graph entities.

```
DocumentCollection graph = await client.CreateDocumentCollectionIfNotExistsAsync(
    UriFactory.CreateDatabaseUri("graphdb"),
    new DocumentCollection { Id = "graphcollz" },
    new RequestOptions { OfferThroughput = 1000 });
```

Serialize vertices and edges to .NET objects

Azure Cosmos DB uses the GraphSON wire format, which defines a JSON schema for vertices, edges, and properties. The Azure Cosmos DB .NET SDK includes JSON.NET as a dependency, and this allows us to serialize/deserialize GraphSON into .NET objects that we can work with in code.

As an example, let's work with a simple social network with four people. We look at how to create `Person` vertices, add `Knows` relationships between them, then query and traverse the graph to find "friend of friend" relationships.

The `Microsoft.Azure.Graphs.Elements` namespace provides `Vertex`, `Edge`, `Property` and `VertexProperty` classes for deserializing GraphSON responses to well-defined .NET objects.

Run Gremlin using CreateGremlinQuery

Gremlin, like SQL, supports read, write, and query operations. For example, the following snippet shows how to create vertices, edges, perform some sample queries using `CreateGremlinQuery<T>`, and asynchronously iterate through these results using `ExecuteNextAsync` and `HasMoreResults.

```
Dictionary<string, string> gremlinQueries = new Dictionary<string, string>
{
    { "Cleanup",        "g.V().drop()" },
    { "AddVertex 1",    "g.addV('person').property('id', 'thomas').property('firstName', 'Thomas').property('age', 44)" },
    { "AddVertex 2",    "g.addV('person').property('id', 'mary').property('firstName', 'Mary').property('lastName', 'Andersen').property('age', 39)" },
    { "AddVertex 3",    "g.addV('person').property('id', 'ben').property('firstName', 'Ben').property('lastName', 'Miller')" },
    { "AddVertex 4",    "g.addV('person').property('id', 'robin').property('firstName', 'Robin').property('lastName', 'Wakefield')" },
    { "AddEdge 1",      "g.V('thomas').addE('knows').to(g.V('mary'))" },
    { "AddEdge 2",      "g.V('thomas').addE('knows').to(g.V('ben'))" },
    { "AddEdge 3",      "g.V('ben').addE('knows').to(g.V('robin'))" },
    { "UpdateVertex",   "g.V('thomas').property('age', 44)" },
    { "CountVertices",  "g.V().count()" },
    { "Filter Range",   "g.V().hasLabel('person').has('age', gt(40))" },
    { "Project",        "g.V().hasLabel('person').values('firstName')" },
    { "Sort",           "g.V().hasLabel('person').order().by('firstName', decr)" },
    { "Traverse",       "g.V('thomas').outE('knows').inV().hasLabel('person')" },
    { "Traverse 2x",    "g.V('thomas').outE('knows').inV().hasLabel('person').outE('knows').inV().hasLabel('person')" },
    { "Loop",           "g.V('thomas').repeat(out()).until(has('id', 'robin')).path()" },
    { "DropEdge",       "g.V('thomas').outE('knows').where(inV().has('id', 'mary')).drop()" },
    { "CountEdges",     "g.E().count()" },
    { "DropVertex",     "g.V('thomas').drop()" },
};

foreach (KeyValuePair<string, string> gremlinQuery in gremlinQueries)
{
    Console.WriteLine($"Running {gremlinQuery.Key}: {gremlinQuery.Value}");

    // The CreateGremlinQuery method extensions allow you to execute Gremlin queries and iterate
    // results asychronously
    IDocumentQuery<dynamic> query = client.CreateGremlinQuery<dynamic>(graph, gremlinQuery.Value);
    while (query.HasMoreResults)
    {
        foreach (dynamic result in await query.ExecuteNextAsync())
        {
            Console.WriteLine($"\t {JsonConvert.SerializeObject(result)}");
        }
    }

    Console.WriteLine();
}
```

Add vertices and edges

Let's look at the Gremlin statements shown in the preceding section more detail. First we some vertices using Gremlin's `addV` method. For example, the following snippet creates a "Thomas Andersen" vertex of type "Person", with properties for first name, last name, and age.

```
// Create a vertex
IDocumentQuery<Vertex> createVertexQuery = client.CreateGremlinQuery<Vertex>(
    graphCollection,
    "g.addV('person').property('firstName', 'Thomas')");

while (createVertexQuery.HasMoreResults)
{
    Vertex thomas = (await create.ExecuteNextAsync<Vertex>()).First();
}
```

Then we create some edges between these vertices using Gremlin's `addE` method.

```
// Add a "knows" edge
IDocumentQuery<Edge> createEdgeQuery = client.CreateGremlinQuery<Edge>(
    graphCollection,
    "g.V('thomas').addE('knows').to(g.V('mary'))");

while (create.HasMoreResults)
{
    Edge thomasKnowsMaryEdge = (await create.ExecuteNextAsync<Edge>()).First();
}
```

We can update an existing vertex by using `properties` step in Gremlin. We skip the call to execute the query via `HasMoreResults` and `ExecuteNextAsync` for the rest of the examples.

```
// Update a vertex
client.CreateGremlinQuery<Vertex>(
    graphCollection,
    "g.V('thomas').property('age', 45)");
```

You can drop edges and vertices using Gremlin's `drop` step. Here's a snippet that shows how to delete a vertex and an edge. Note that dropping a vertex performs a cascading delete of the associated edges.

```
// Drop an edge
client.CreateGremlinQuery(graphCollection, "g.E('thomasKnowsRobin').drop()");

// Drop a vertex
client.CreateGremlinQuery(graphCollection, "g.V('robin').drop()");
```

Query the graph

You can perform queries and traversals also using Gremlin. For example, the following snippet shows how to count the number of vertices in the graph:

```
// Run a query to count vertices
IDocumentQuery<int> countQuery = client.CreateGremlinQuery<int>(graphCollection, "g.V().count()");
```

You can perform filters using Gremlin's `has` and `hasLabel` steps, and combine them using `and`, `or`, and `not` to build more complex filters:

```
// Run a query with filter
IDocumentQuery<Vertex> personsByAge = client.CreateGremlinQuery<Vertex>(
  graphCollection,
  "g.V().hasLabel('person').has('age', gt(40))");
```

You can project certain properties in the query results using the `values` step:

```
// Run a query with projection
IDocumentQuery<string> firstNames = client.CreateGremlinQuery<string>(
  graphCollection,
  $"g.V().hasLabel('person').values('firstName')");
```

So far, we've only seen query operators that work in any database. Graphs are fast and efficient for traversal operations when you need to navigate to related edges and vertices. Let's find all friends of Thomas. We do this by using Gremlin's `outE` step to find all the out-edges from Thomas, then traversing to the in-vertices from those edges using Gremlin's `inV` step:

```
// Run a traversal (find friends of Thomas)
IDocumentQuery<Vertex> friendsOfThomas = client.CreateGremlinQuery<Vertex>(
  graphCollection,
  "g.V('thomas').outE('knows').inV().hasLabel('person')");
```

The next query performs two hops to find all of Thomas' "friends of friends", by calling `outE` and `inV` two times.

```
// Run a traversal (find friends of friends of Thomas)
IDocumentQuery<Vertex> friendsOfFriendsOfThomas = client.CreateGremlinQuery<Vertex>(
  graphCollection,
  "g.V('thomas').outE('knows').inV().hasLabel('person').outE('knows').inV().hasLabel('person')");
```

You can build more complex queries and implement powerful graph traversal logic using Gremlin, including mixing filter expressions, performing looping using the `loop` step, and implementing conditional navigation using the `choose` step. Learn more about what you can do with Gremlin support!

That's it, this Azure Cosmos DB tutorial is complete!

Clean up resources

If you're not going to continue to use this app, use the following steps to delete all resources created by this tutorial in the Azure portal.

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next Steps

In this tutorial, you've done the following:

- Created an Azure Cosmos DB account
- Created a graph database and container
- Serialized vertices and edges to .NET objects
- Added vertices and edges
- Queried the graph using Gremlin

You can now build more complex queries and implement powerful graph traversal logic using Gremlin.

Query using Gremlin

How to import data into Azure Cosmos DB for the DocumentDB API?

6/12/2017 • 23 min to read • Edit Online

This tutorial provides instructions on using the Azure Cosmos DB: DocumentDB API Data Migration tool, which can import data from various sources, including JSON files, CSV files, SQL, MongoDB, Azure Table storage, Amazon DynamoDB and Azure Cosmos DB DocumentDB API collections into collections for use with Azure Cosmos DB and the DocumentDB API. The Data Migration tool can also be used when migrating from a single partition collection to a multi-partition collection for the DocumentDB API.

The Data Migration tool only works when importing data into Azure Cosmos DB for use with the DocumentDB API. Importing data for use with the Table API or Graph API is not supported at this time.

To import data for use with the MongoDB API, see Azure Cosmos DB: How to migrate data for the MongoDB API?.

This tutorial covers the following tasks:

- Installing the Data Migration tool
- Importing data from different data sources
- Exporting from Azure Cosmos DB to JSON

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 Data Migration tool

The Data Migration tool is an open source solution that imports data to Azure Cosmos DB from a variety of sources, including:

- JSON files
- MongoDB
- SQL Server
- CSV files
- Azure Table storage
- Amazon DynamoDB
- HBase
- Azure Cosmos DB 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.

Install the 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

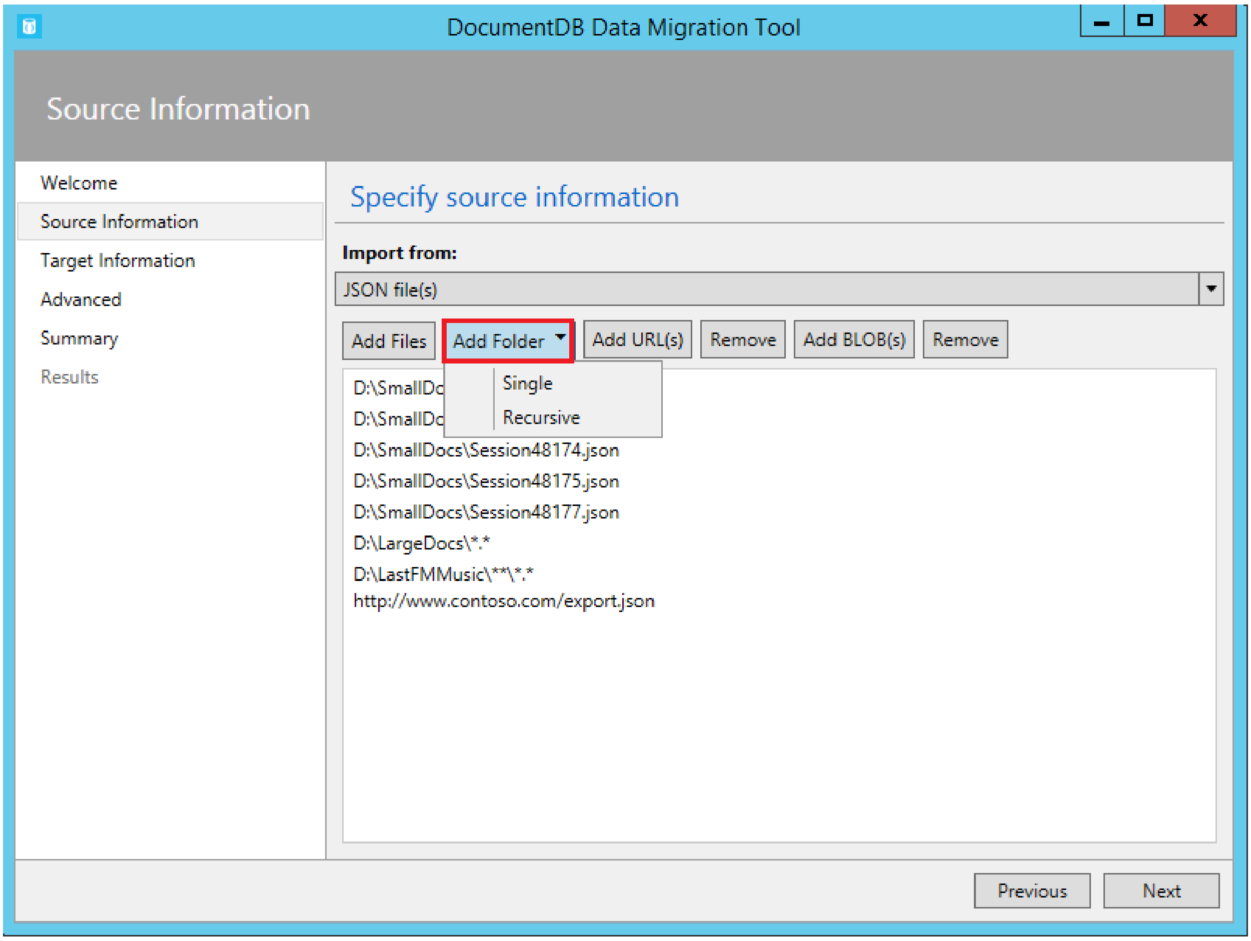
Import data

Once you've installed the tool, it's time to import your data. What kind of data do you want to import?

- JSON files
- MongoDB
- MongoDB Export files
- SQL Server
- CSV files
- Azure Table storage
- Amazon DynamoDB
- Blob
- Azure Cosmos DB collections
- HBase
- Azure Cosmos DB bulk import
- Azure Cosmos DB sequential record import

To 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.



Import from: JSON file(s)

```
D:\SmallDc
D:\SmallDc
D:\SmallDocs\Session48174.json
D:\SmallDocs\Session48175.json
D:\SmallDocs\Session48177.json
D:\LargeDocs\*.*
D:\LastFMMusic\**\*.*
http://www.contoso.com/export.json
```

Here are some command line samples to import JSON files:

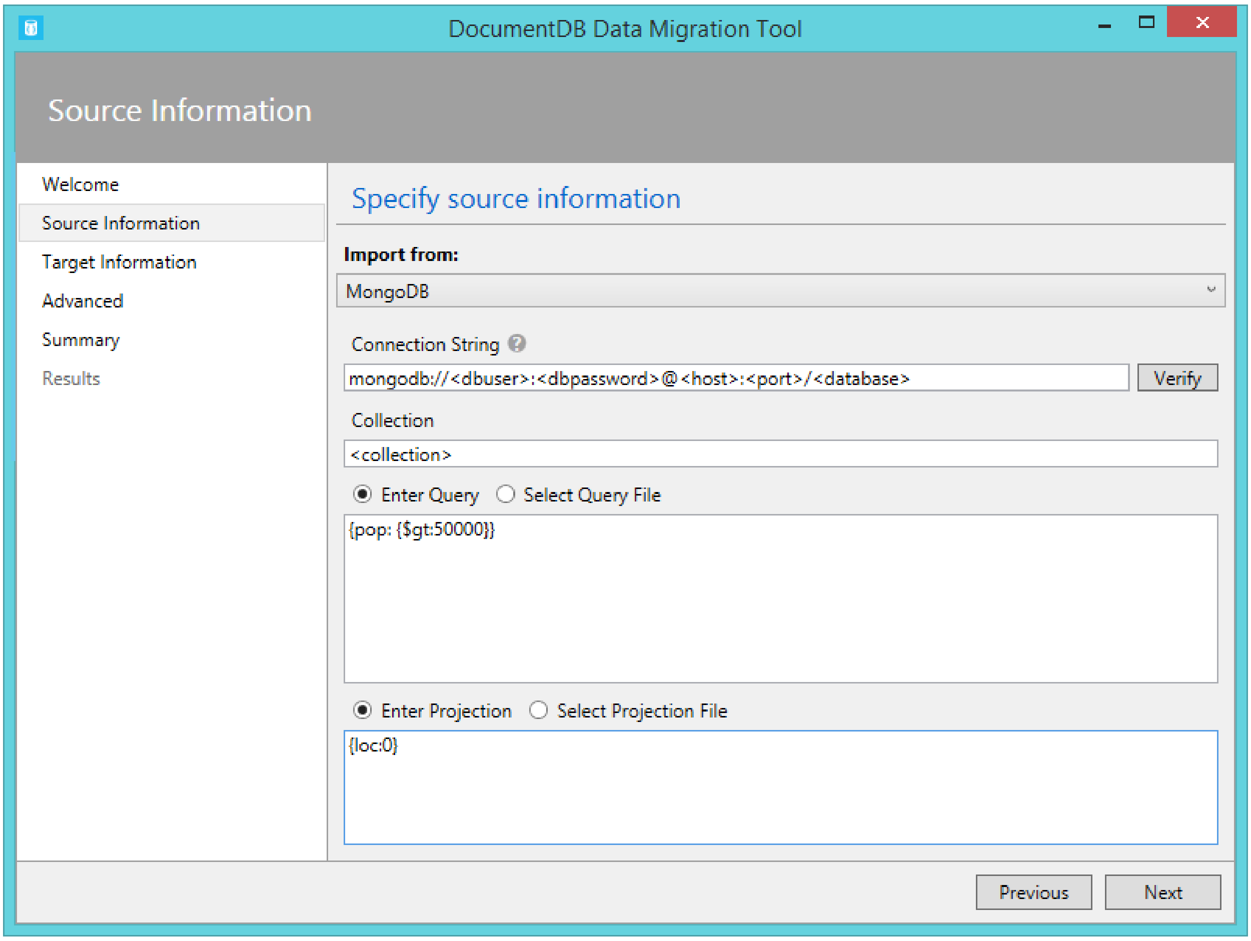
```
#Import a single JSON file
dt.exe /s:JsonFile /s.Files:.\Sessions.json /t:CosmosDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=
<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:Sessions /t.CollectionThroughput:2500

#Import a directory of JSON files
dt.exe /s:JsonFile /s.Files:C:\TESessions\*.json /t:CosmosDBBulk /t.ConnectionString:" AccountEndpoint=<CosmosDB
Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB 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:CosmosDBBulk /t.ConnectionString:" AccountEndpoint=<CosmosDB
Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB 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:\TESessions\Session48174
.json;C:\TESessions\Session48175.json;C:\TESessions\Session48177.json /t:CosmosDBBulk /t.ConnectionString:"AccountEndpoint=
<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB 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:CosmosDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB
Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:comp[1-4] /t.PartitionKey:name
/t.CollectionThroughput:2500
```

To import from MongoDB

IMPORTANT

If you are importing to an Azure Cosmos DB account with Support for MongoDB, follow these instructions.

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.



Import from: MongoDB

Connection String

mongodb://<dbuser>:<dbpassword>@<host>:<port>/<database>

Collection

<collection>

● Enter Query ○ Select Query File

{pop: {$gt:50000}}

● Enter Projection ○ Select Projection File

{loc:0}

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:

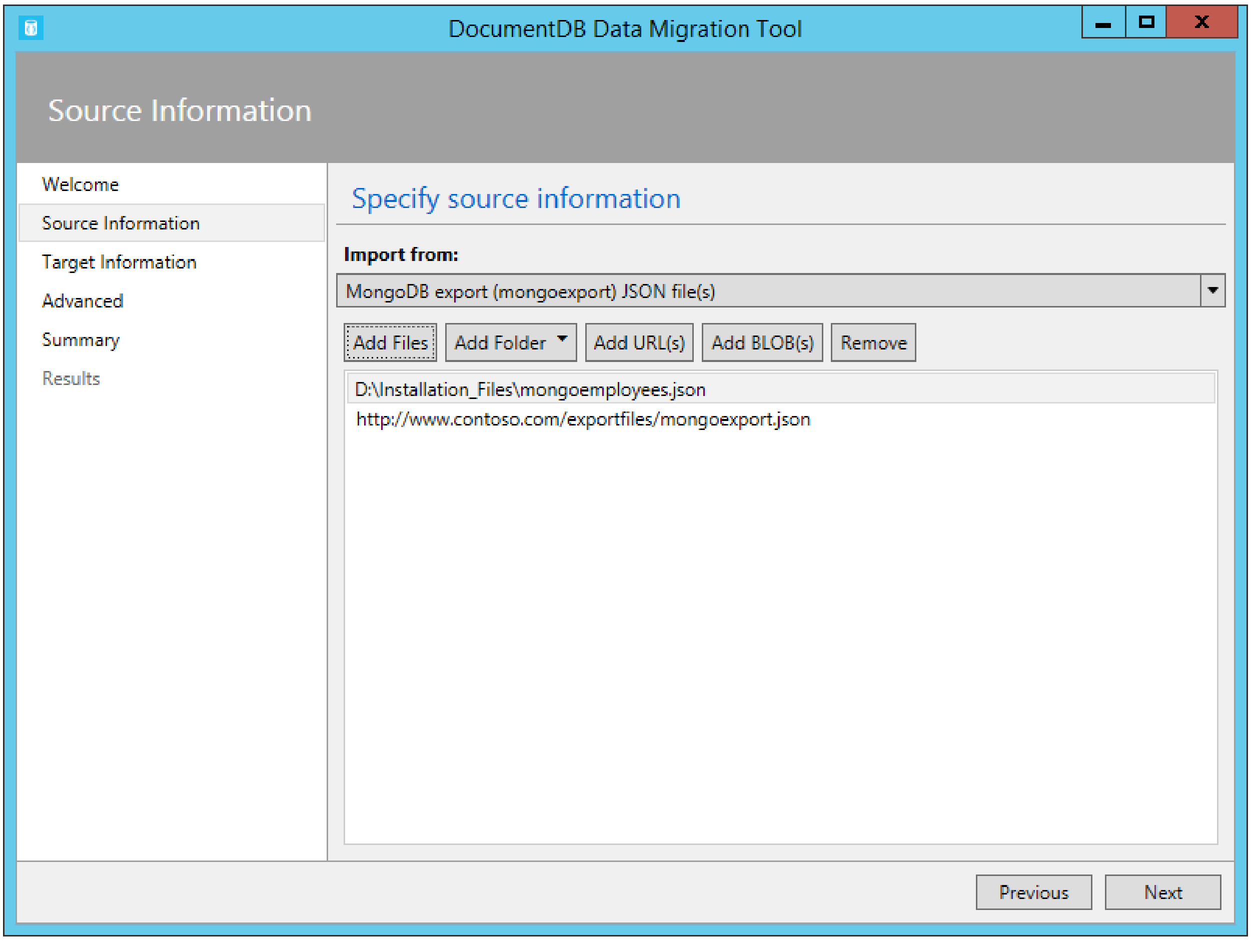
```
#Import all documents from a MongoDB collection
dt.exe /s:MongoDB /s.ConnectionString:mongodb://<dbuser>:<dbpassword>@<host>:<port>/<database> /s.Collection:zips
/t:CosmosDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB
Database>;" /t.Collection:BulkZips /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:CosmosDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=
<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:BulkZipsTransform /t.IdField:_id/t.CollectionThroughput:2500
```

To import MongoDB export files

IMPORTANT

If you are importing to an Azure Cosmos DB account with support for MongoDB, follow these instructions.

The MongoDB export JSON file source importer option allows you to import one or more JSON files produced from the mongoexport utility.



DocumentDB Data Migration Tool

Source Information

Specify source information

Import from: MongoDB export (mongoexport) JSON file(s)

D:\Installation_Files\mongoemployees.json
http://www.contoso.com/exportfiles/mongoexport.json

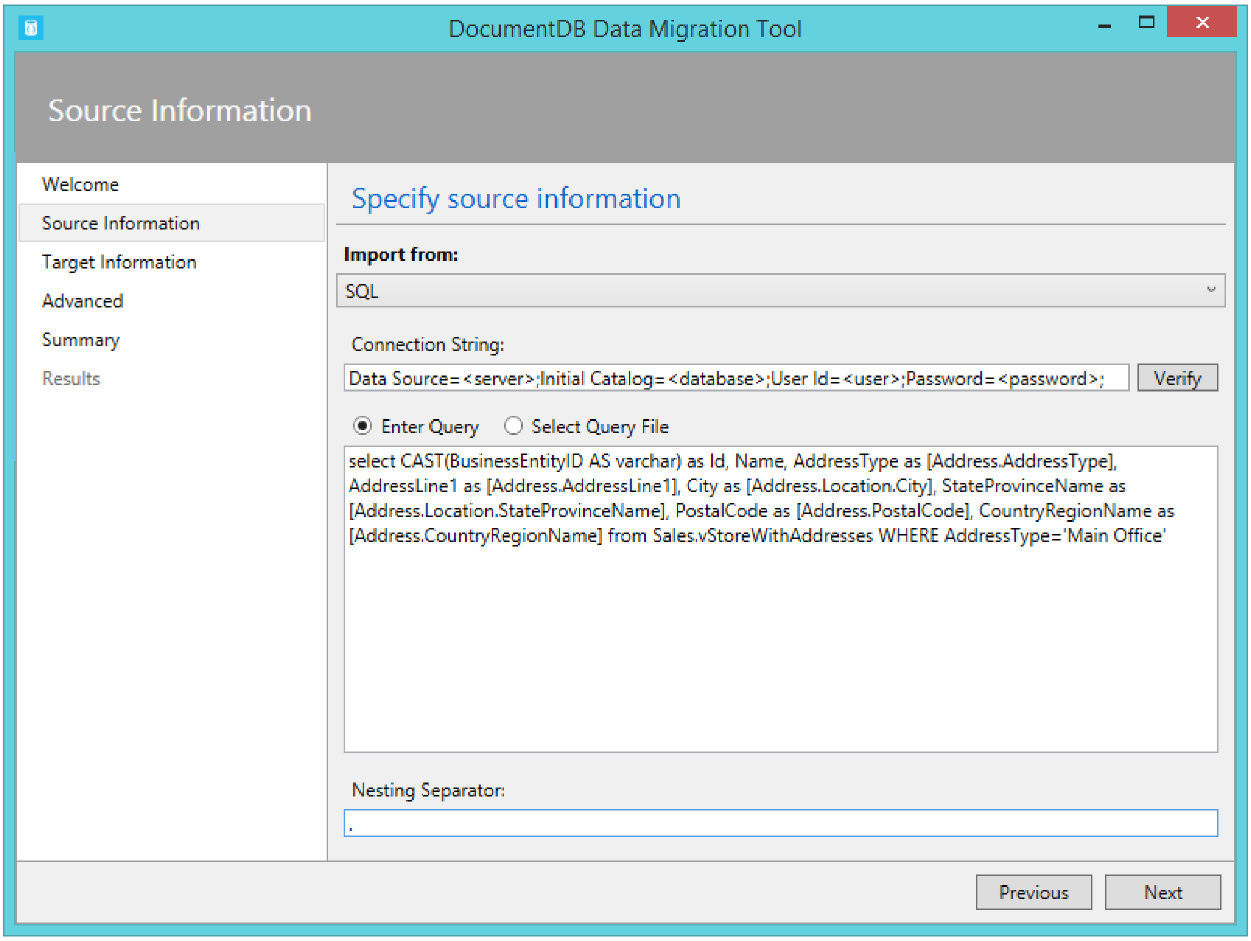
When adding folders that contain MongoDB export JSON files for import, you have the option of recursively searching for files in subfolders.

Here is a command line sample to import from MongoDB export JSON files:

```
dt.exe /s:MongoDBExport /s.Files:D:\mongoemployees.json /t:CosmosDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:employees /t.IdField:_id /t.Dates:Epoch /t.CollectionThroughput:2500
```

To 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).



Specify source information

Import from: SQL

Connection String:

Data Source=<server>;Initial Catalog=<database>;User Id=<user>;Password=<password>;

● Enter Query ○ Select Query File

```
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'
```

Nesting Separator:

.

The format of the connection string is the standard SQL connection string format.

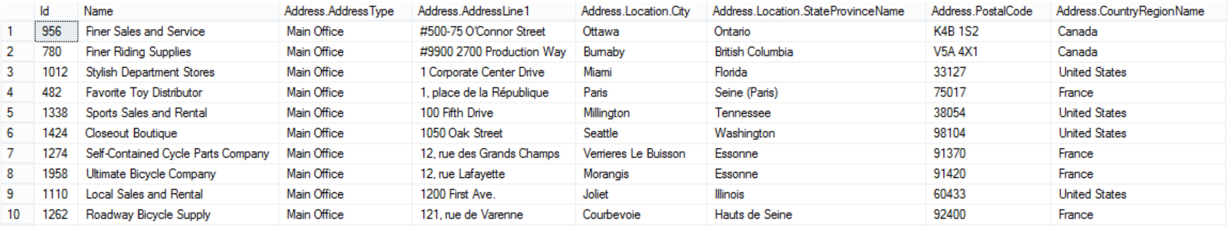
NOTE

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	Bumaby	British Columbia	V5A 4X1	Canada
3	1012	Stylish Department Stores	Main Office	1 Corporate Center Drive	Miami	Florida	33127	United States
4	482	Favorite 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	Verrieres 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 Azure Cosmos DB:

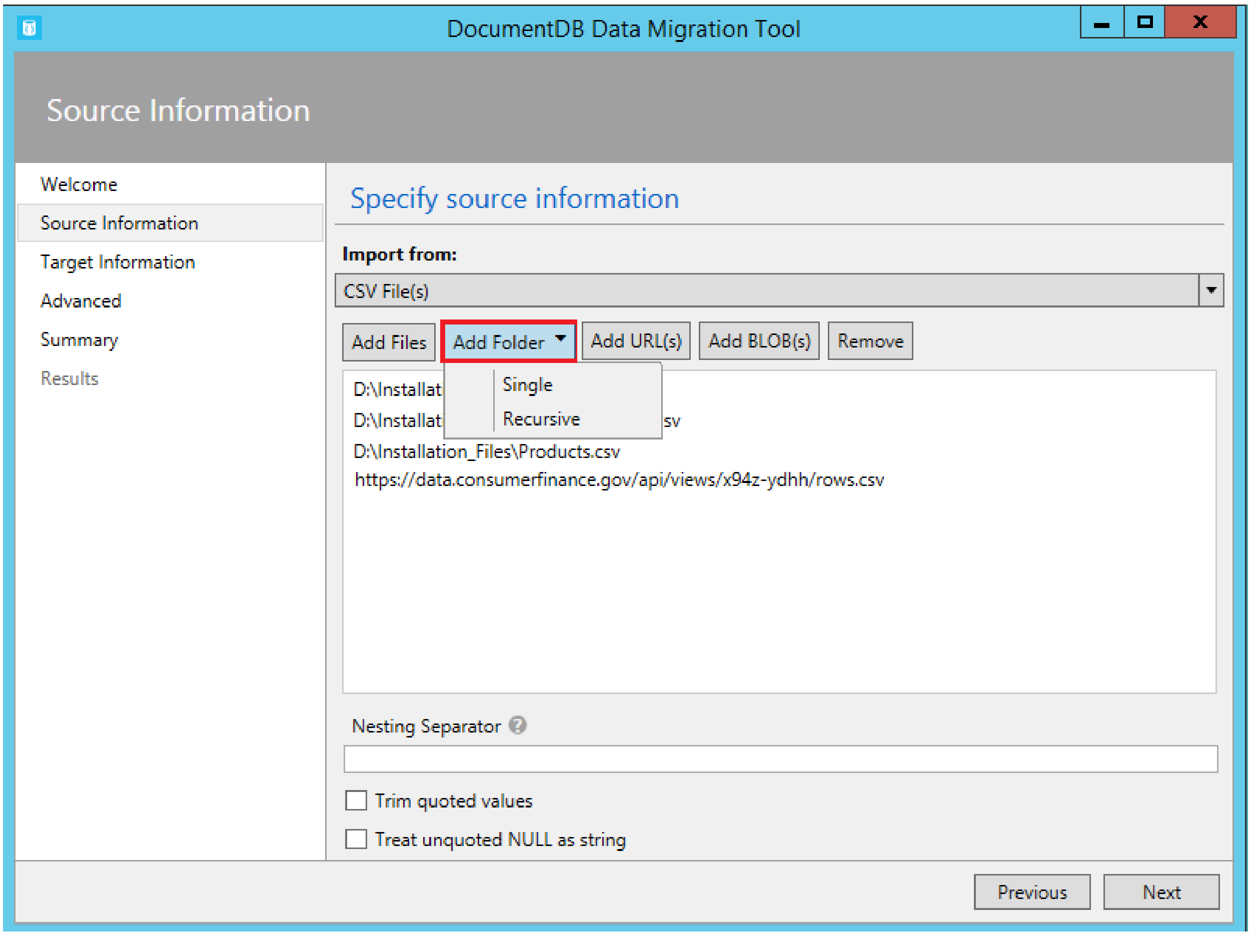
{ "id": "956", "Name": "Finer Sales and Service", "Address": { "AddressType": "Main Office", "AddressLine1": "#500-75 O'Connor Street", "Location": { "City": "Ottawa", "StateProvinceName": "Ontario" }, "PostalCode": "K4B 1S2", "CountryRegionName": "Canada" } }

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:CosmosDBBulk /t.ConnectionString:" AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB
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:CosmosDBBulk /t.ConnectionString:" AccountEndpoint=<CosmosDB
Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:StoresSub /t.IdField:Id
/t.CollectionThroughput:2500
```

To import CSV files and 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.



Import from:

CSV File(s)

D:\Installat

D:\Installat sv

D:\Installation_Files\Products.csv

https://data.consumerfinance.gov/api/views/x94z-ydhh/rows.csv

□ Trim quoted values

□ Treat unquoted NULL as string

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 Azure Cosmos DB:

{ "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:

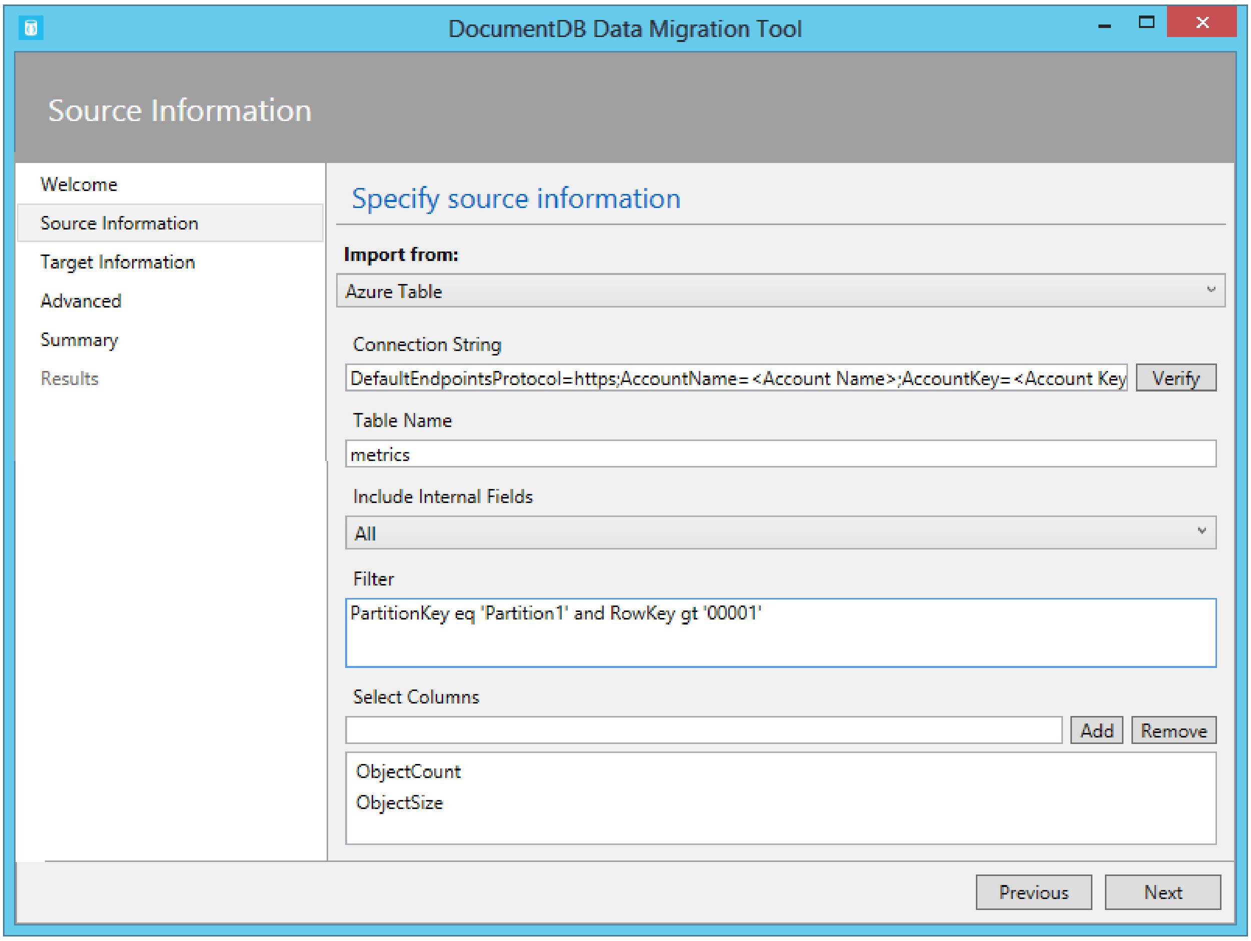
1. By default, unquoted values are always trimmed for tabs and spaces, while quoted values are preserved as-is. This behavior can be overridden with the Trim quoted values checkbox or the /s.TrimQuoted command line option.
2. By default, an unquoted null is treated as a null value. This behavior can 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:CosmosDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=
<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:Employees /t.IdField:EntityID /t.CollectionThroughput:2500
```

To 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. Note that you cannot use the Data Migration tool to import Azure Table storage data into Azure Cosmos DB for use with the Table API. Only importing to Azure Cosmos DB for use with the DocumentDB API is supported at this time.



Import from: Azure Table

Connection String: DefaultEndpointsProtocol=https;AccountName=<Account Name>;AccountKey=<Account Key

Table Name: metrics

Include Internal Fields: All

Filter: PartitionKey eq 'Partition1' and RowKey gt '00001'

Select Columns:
ObjectCount
ObjectSize

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:

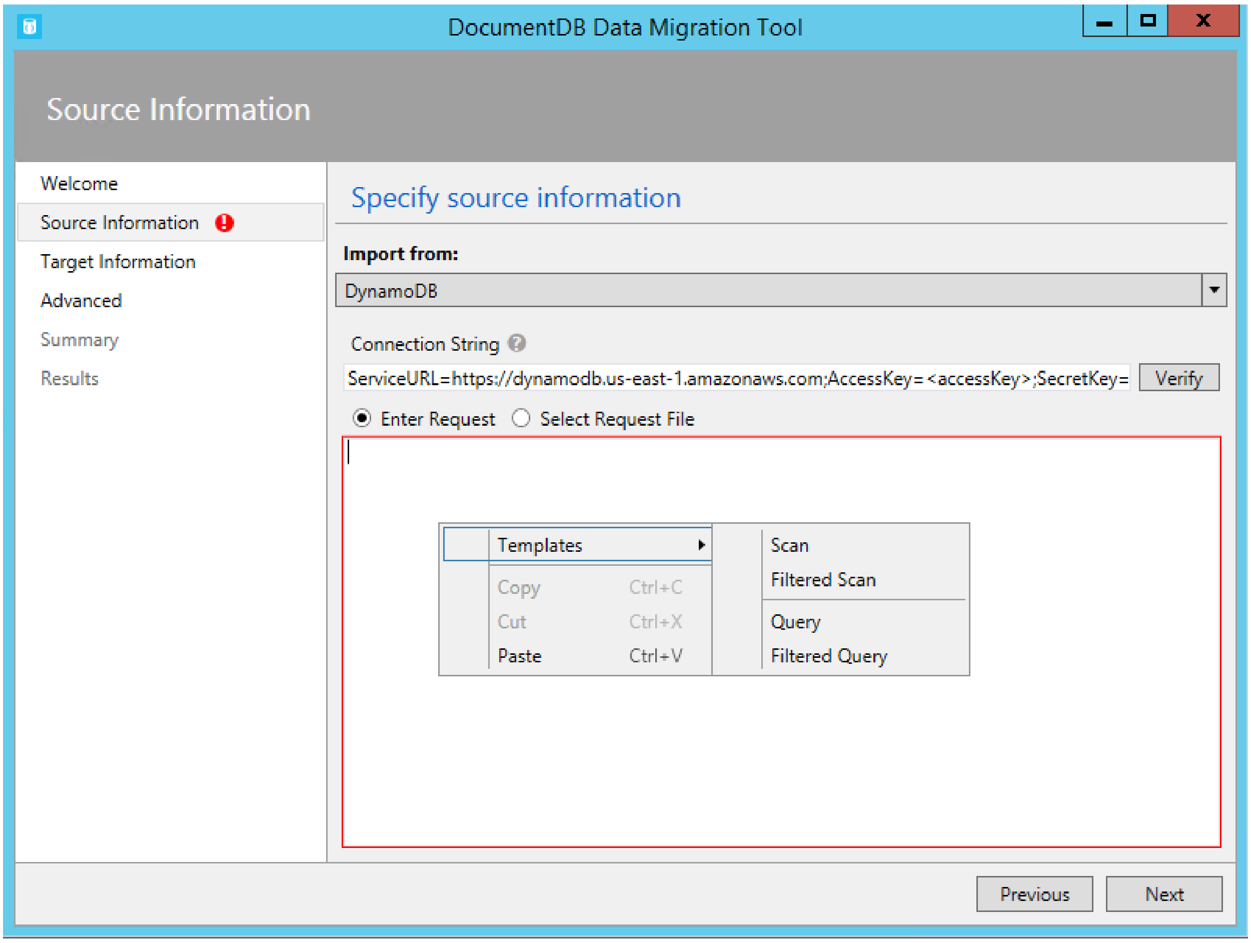
1. Include Internal Fields
   a. All - Include all internal fields (PartitionKey, RowKey, and Timestamp)
   b. None - Exclude all internal fields
   c. RowKey - Only include the RowKey field
2. Select Columns
   a. Azure Table storage filters do not support projections. If you want to only import specific 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:CosmosDBBulk /t.ConnectionString:" AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:metrics /t.CollectionThroughput:2500
```

To 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.

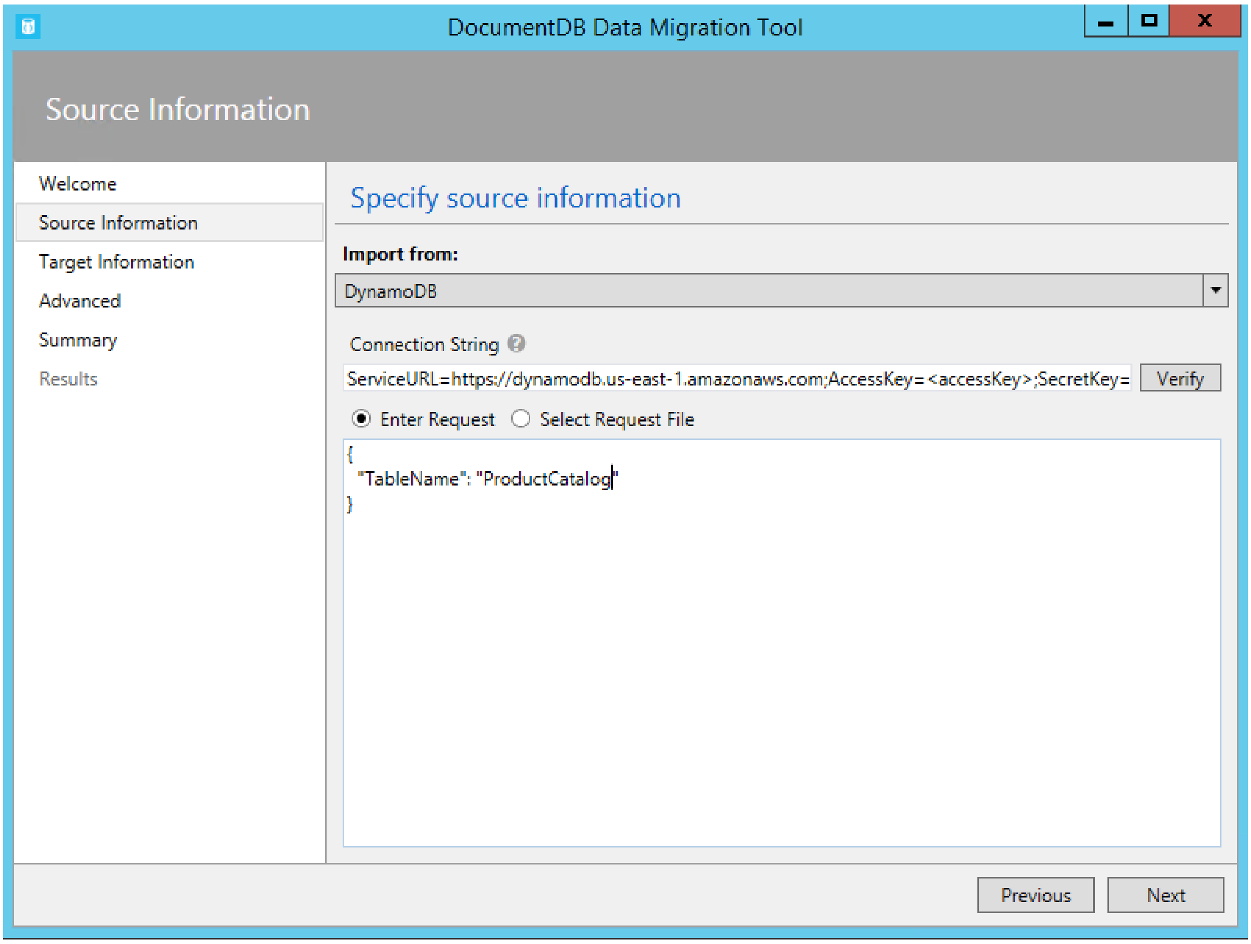


Import from: DynamoDB

Connection String

ServiceURL=https://dynamodb.us-east-1.amazonaws.com;AccessKey=<accessKey>;SecretKey=

● Enter Request ○ Select Request File



Import from: DynamoDB

Connection String

ServiceURL=https://dynamodb.us-east-1.amazonaws.com;AccessKey=<accessKey>;SecretKey=

● Enter Request ○ Select Request File

```
{
 "TableName": "ProductCatalog"
}
```

The format of the Amazon DynamoDB connection string is:

```
ServiceURL=<Service Address>;AccessKey=<Access Key>;SecretKey=<Secret Key>;
```

NOTE

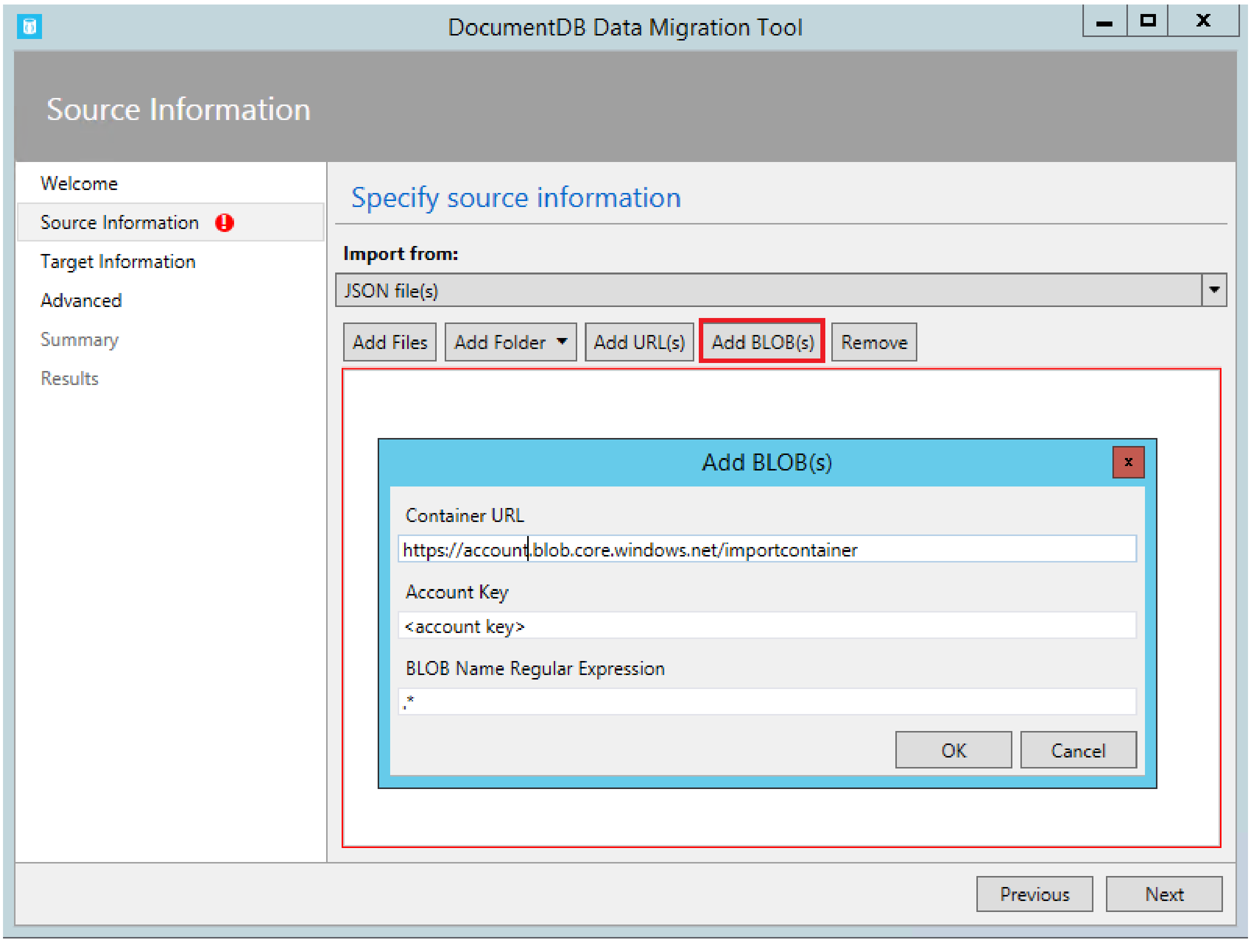
Use the Verify command to ensure that the Amazon DynamoDB instance specified in the connection string field can be accessed.

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=<Azure
Cosmos DB Endpoint>;AccountKey=<Azure Cosmos DB Key>;Database=<Azure Cosmos DB Database>;" /t.Collection:catalogCollection
/t.CollectionThroughput:2500
```

To 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.



Import from: JSON file(s)

Container URL: https://account.blob.core.windows.net/importcontainer

Account Key: <account key>

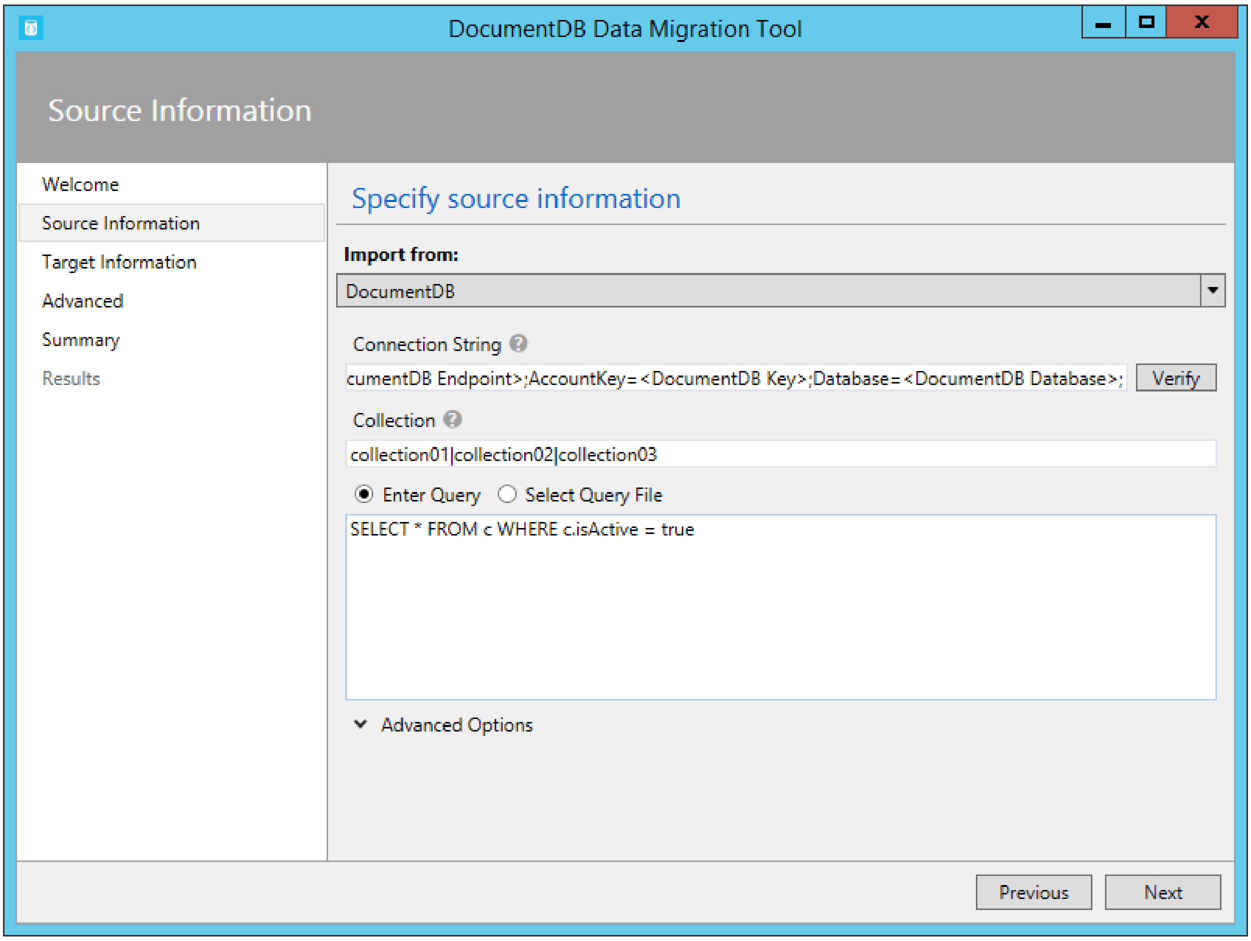
BLOB Name Regular Expression: .*

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:CosmosDBBulk
/t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;"
/t.Collection:doctest
```

To import from an Azure Cosmos DB DocumentDB API collection

The Azure Cosmos DB source importer option allows you to import data from one or more Azure Cosmos DB collections and optionally filter documents using a query.



Import from: DocumentDB

Connection String

cumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;

Collection

collection01|collection02|collection03

● Enter Query ○ Select Query File

```
SELECT * FROM c WHERE c.isActive = true
```

The format of the Azure Cosmos DB connection string is:

```
AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;
```

The Azure Cosmos DB account connection string can be retrieved from the Keys blade of the Azure portal, as described in How to manage an Azure Cosmos DB account, however the name of the database needs to be appended to the connection string in the following format:

```
Database=<CosmosDB Database>;
```

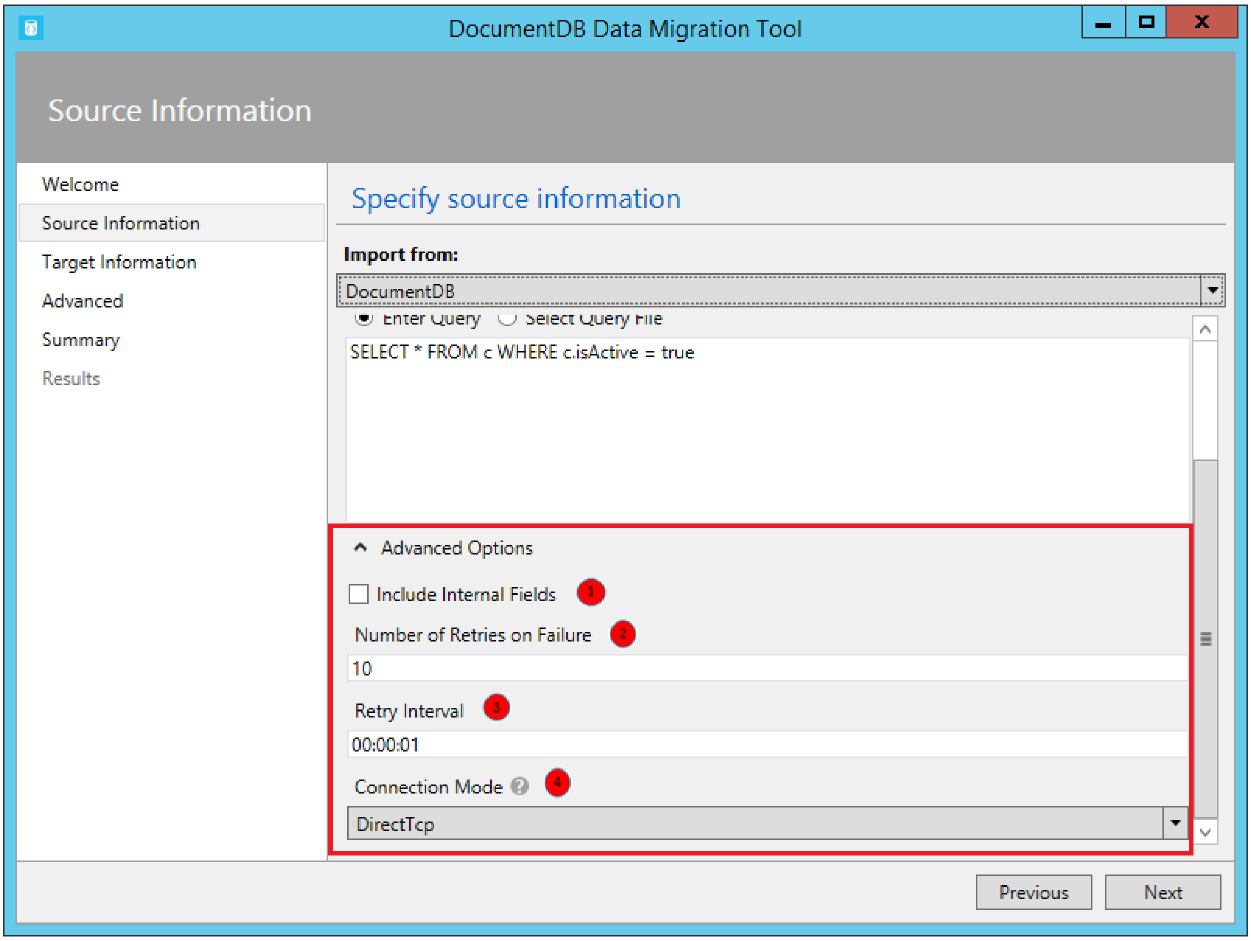
> NOTE
>
> Use the Verify command to ensure that the Azure Cosmos DB instance specified in the connection string field can be accessed.

To import from a single Azure Cosmos DB collection, enter the name of the collection from which data will be imported. To import from multiple Azure Cosmos DB 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 Azure Cosmos DB source importer option has the following advanced options:

1. Include Internal Fields: Specifies whether or not to include Azure Cosmos DB 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 Azure Cosmos DB in case of transient failures (e.g. network connectivity interruption).
3. Retry Interval: Specifies how long to wait between retrying the connection to Azure Cosmos DB in case of transient failures (e.g. network connectivity interruption).
4. Connection Mode: Specifies the connection mode to use with Azure Cosmos DB. 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.



Import from: DocumentDB

● Enter Query ○ Select Query File

SELECT * FROM c WHERE c.isActive = true

Advanced Options

□ Include Internal Fields

Number of Retries on Failure: 10

Retry Interval: 00:00:01

Connection Mode: DirectTcp

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 Azure Cosmos DB:

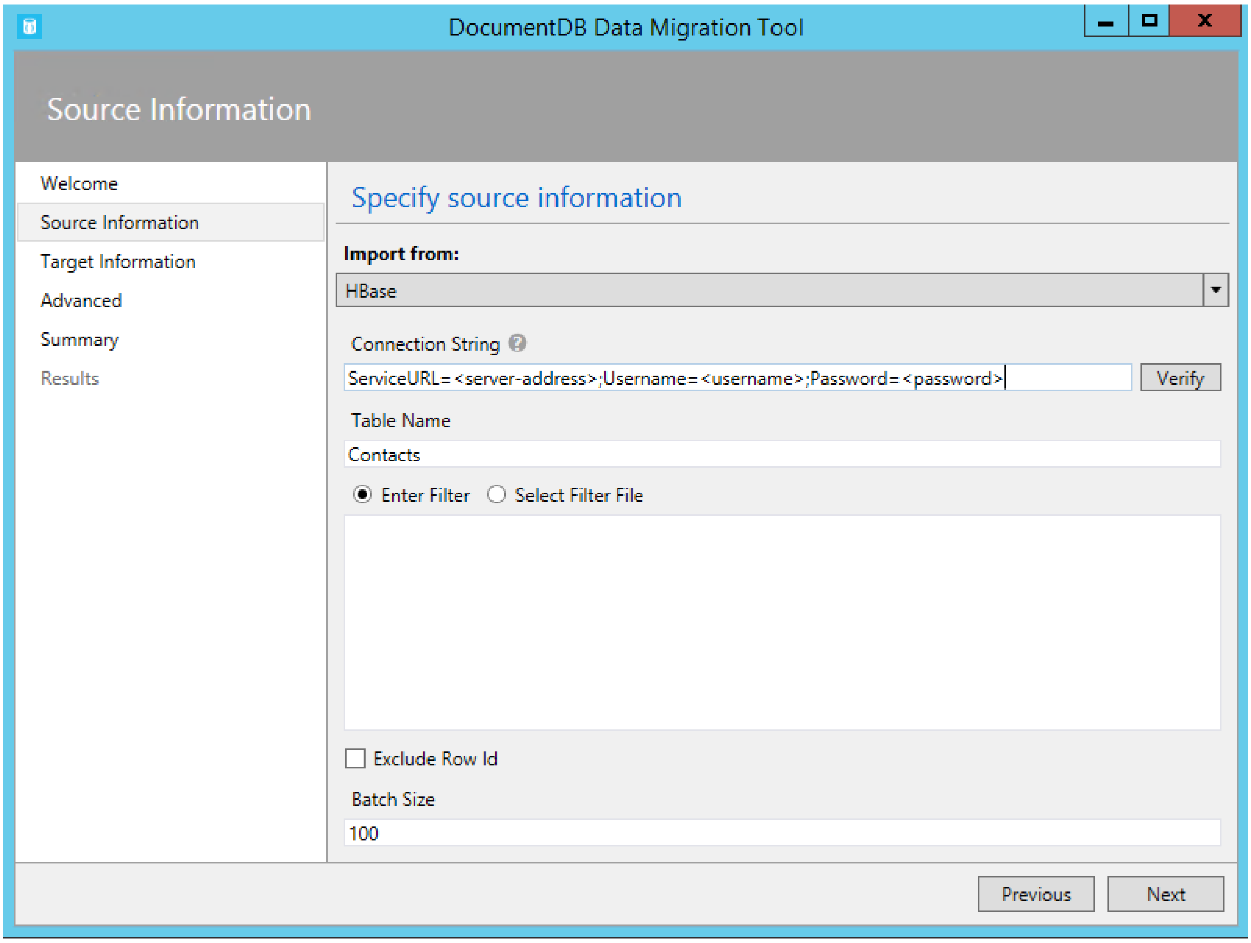
```
#Migrate data from one Azure Cosmos DB collection to another Azure Cosmos DB collections
dt.exe /s:CosmosDB /s.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=
<CosmosDB Database>;" /s.Collection:TEColl /t:CosmosDBBulk /t.ConnectionString:" AccountEndpoint=<CosmosDB
Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:TESessions /t.CollectionThroughput:2500

#Migrate data from multiple Azure Cosmos DB collections to a single Azure Cosmos DB collection
dt.exe /s:CosmosDB /s.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=
<CosmosDB Database>;" /s.Collection:comp1|comp2|comp3|comp4 /t:CosmosDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB
Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:singleCollection /t.CollectionThroughput:2500

#Export an Azure Cosmos DB collection to a JSON file
dt.exe /s:CosmosDB /s.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=
<CosmosDB Database>;" /s.Collection:StoresSub /t:JsonFile /t.File:StoresExport.json /t.Overwrite /t.CollectionThroughput:2500
```

TIP

The Azure Cosmos DB Data Import Tool also supports import of data from the Azure Cosmos DB Emulator. When importing data from a local emulator, set the endpoint to `https://localhost:<port>`.

To 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.



Specify source information

Import from: HBase

Connection String

ServiceURL=<server-address>;Username=<username>;Password=<password>

Table Name

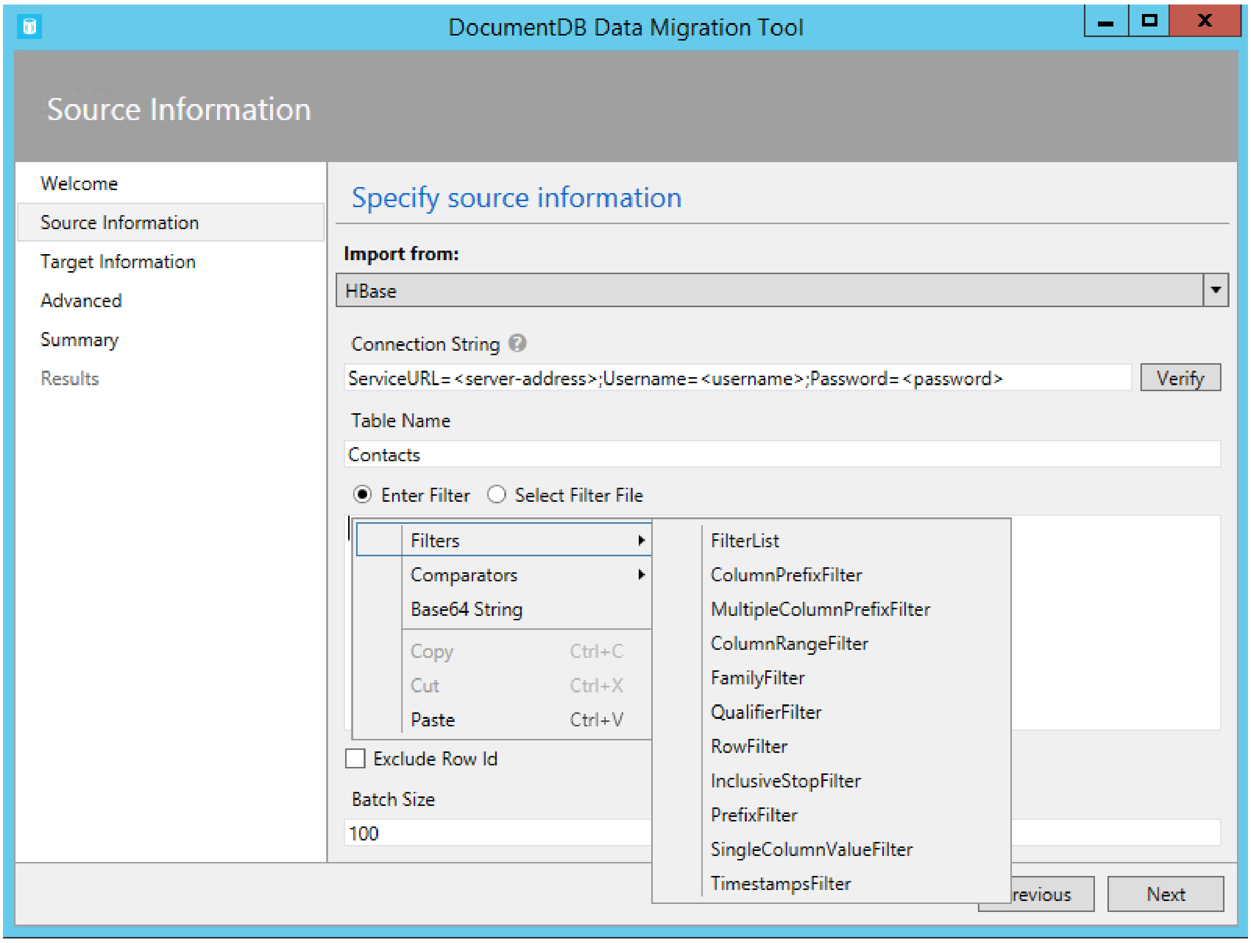
Contacts

● Enter Filter ○ Select Filter File

□ Exclude Row Id

Batch Size

100



Import from: HBase

Connection String
ServiceURL=<server-address>;Username=<username>;Password=<password>

Table Name
Contacts

● Enter Filter ○ Select Filter File

□ Exclude Row Id

Batch Size
100

The format of the HBase Stargate connection string is:

```
ServiceURL=<server-address>;Username=<username>;Password=<password>
```

NOTE

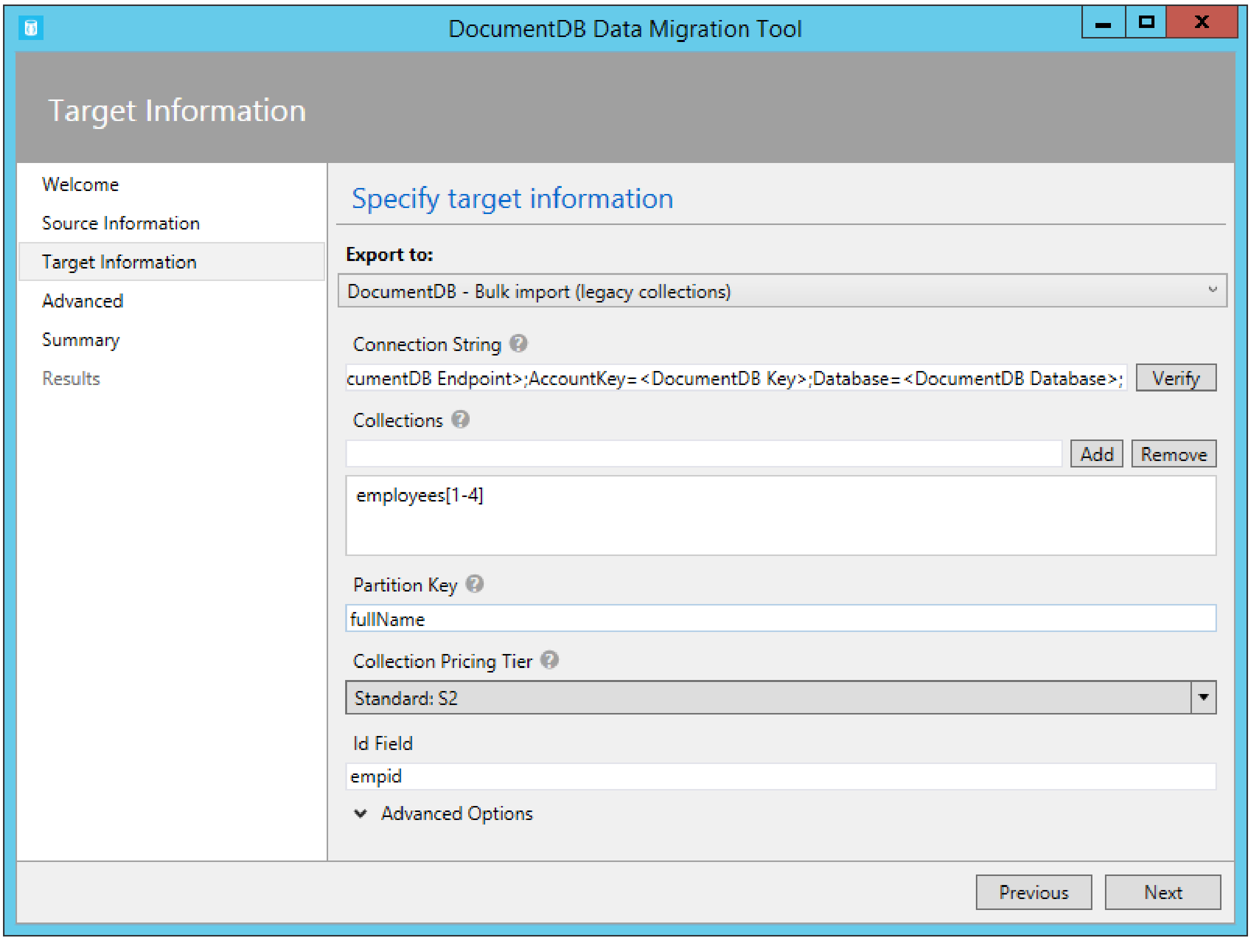
Use the Verify command to ensure that the HBase instance specified in the connection string field can be accessed.

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:CosmosDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB
Database>;" /t.Collection:hbaseimport
```

To import to the DocumentDB API (Bulk Import)

The Azure Cosmos DB Bulk importer allows you to import from any of the available source options, using an Azure Cosmos DB stored procedure for efficiency. The tool supports import to one single-partitioned Azure Cosmos DB collection, as well as sharded import whereby data is partitioned across multiple single-partitioned Azure Cosmos DB collections. For more information about partitioning data, see Partitioning and scaling in Azure Cosmos DB. The tool will create, execute, and then delete the stored procedure from the target collection(s).



Target Information

Specify target information

Export to: DocumentDB - Bulk import (legacy collections)

Connection String

cumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;

Collections

employees[1-4]

Partition Key

fullName

Collection Pricing Tier

Standard: S2

Id Field

empid

The format of the Azure Cosmos DB connection string is:

```
AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;
```

The Azure Cosmos DB account connection string can be retrieved from the Keys blade of the Azure portal, as described in How to manage an Azure Cosmos DB account, however the name of the database needs to be appended to the connection string in the following format:

```
Database=<CosmosDB Database>;
```

NOTE

Use the Verify command to ensure that the Azure Cosmos DB 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 Azure Cosmos DB.

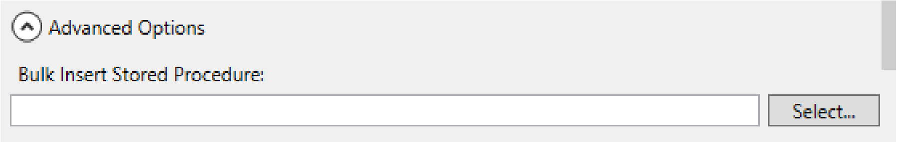
NOTE

The performance throughput setting only applies to collection creation. If the specified collection already exists, its throughput will not be modified.

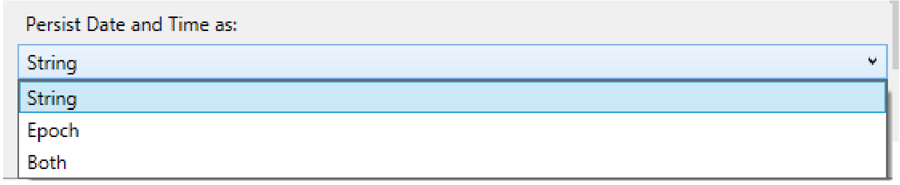
When importing to multiple collections, the import tool supports hash based sharding. In this scenario, specify the document property you wish to use as the Partition Key (if Partition Key is left blank, documents will be sharded randomly across the target collections).

You may optionally specify which field in the import source should be used as the Azure Cosmos DB 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:



Additionally, when importing date types (e.g. from SQL Server or MongoDB), you can choose between three import options:

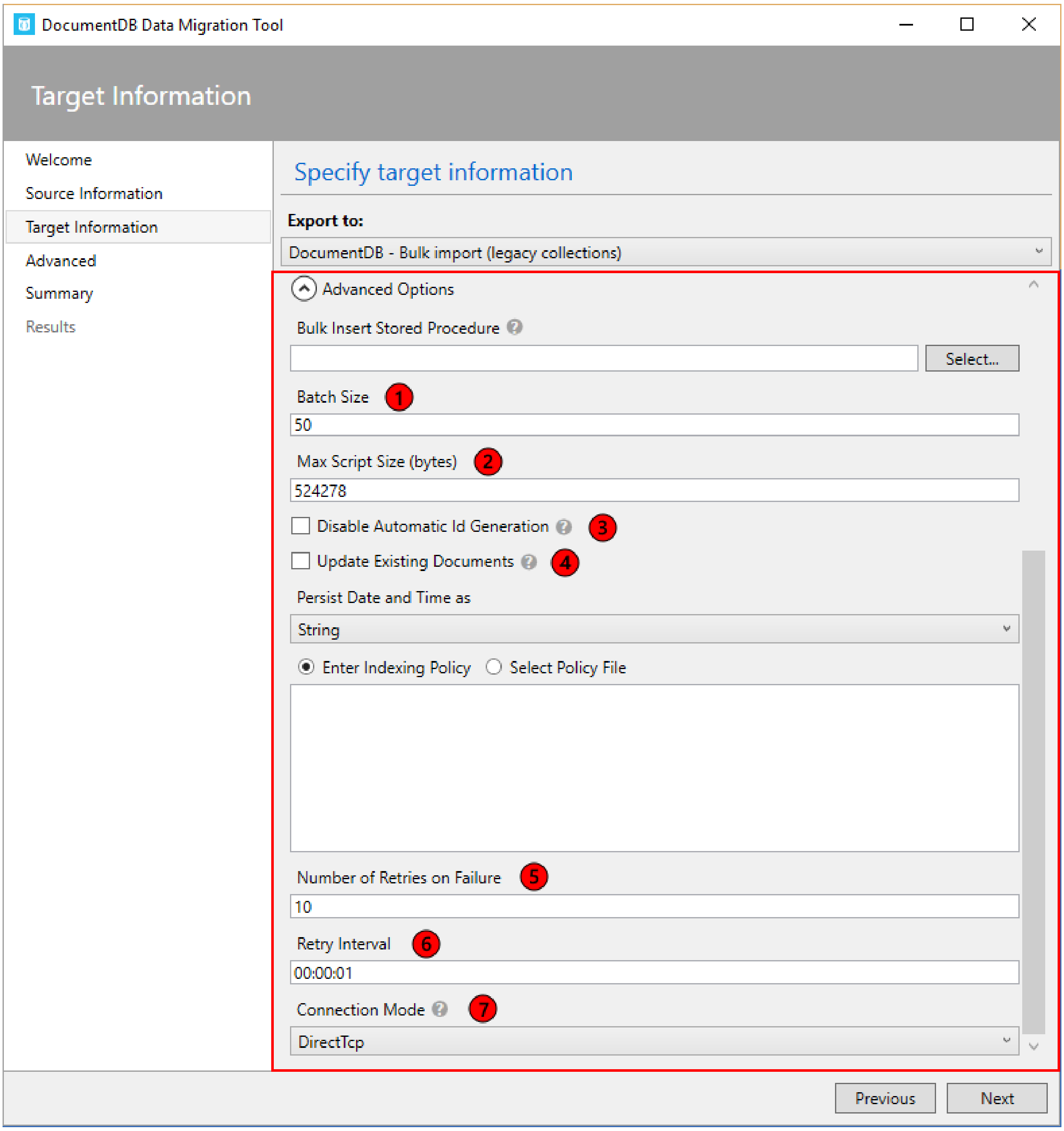


- String: Persist as a string value
- Epoch: Persist as an Epoch number value
- Both: Persist both string and Epoch number values. This option will create a subdocument, for example: "date_joined": { "Value": "2013-10-21T21:17:25.2410000Z", "Epoch": 1382390245 }

The Azure Cosmos DB 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 Azure Cosmos DB in case of transient failures (e.g. network connectivity interruption).
6. Retry Interval: Specifies how long to wait between retrying the connection to Azure Cosmos DB in case of transient failures (e.g. network connectivity interruption).
7. Connection Mode: Specifies the connection mode to use with Azure Cosmos DB. 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

Specify target information

Export to: DocumentDB - Bulk import (legacy collections)

Advanced Options

Bulk Insert Stored Procedure: (empty) [Select...]

1 Batch Size: 50

2 Max Script Size (bytes): 524278

3 □ Disable Automatic Id Generation

4 □ Update Existing Documents

Persist Date and Time as: String

● Enter Indexing Policy ○ Select Policy File

5 Number of Retries on Failure: 10

6 Retry Interval: 00:00:01

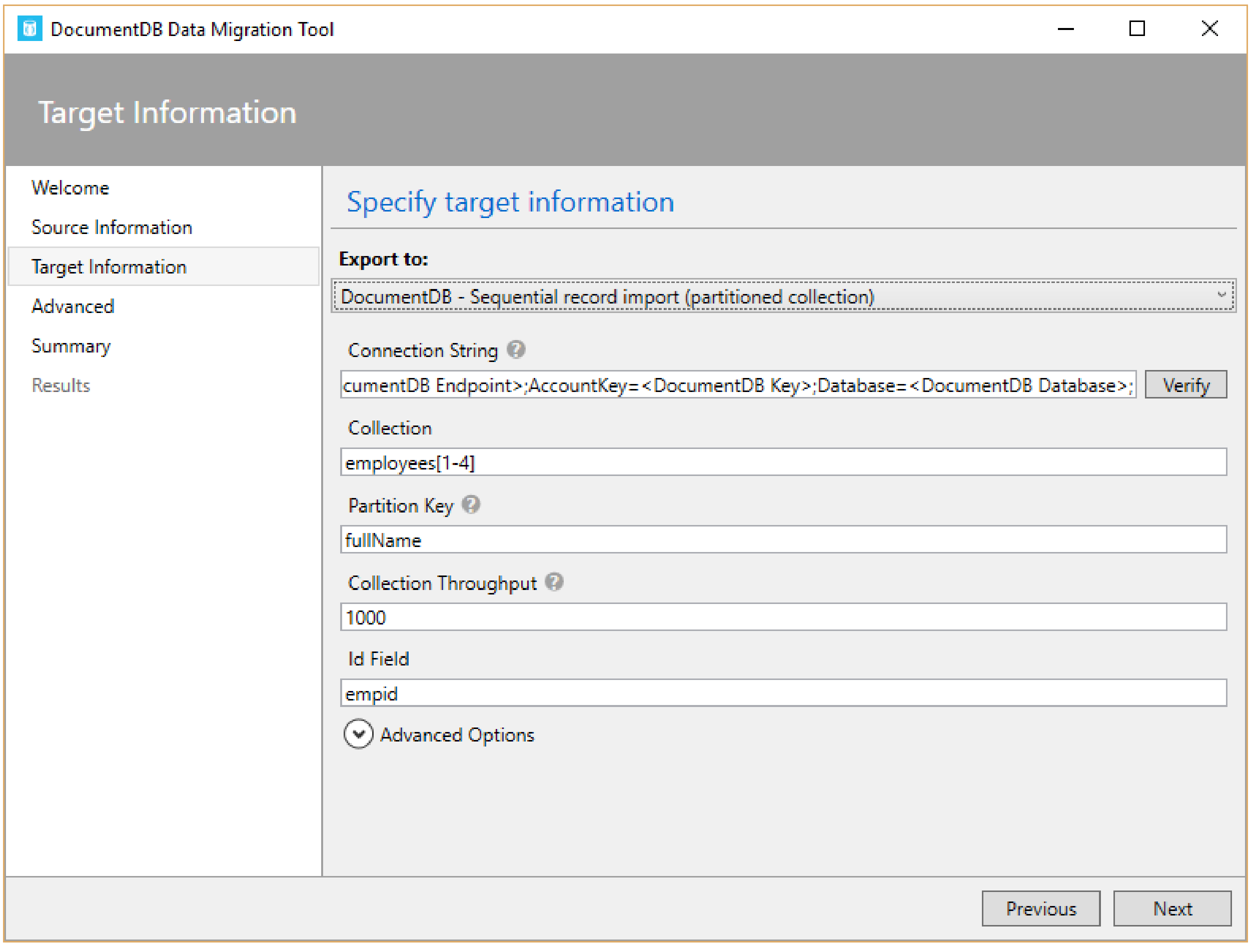
7 Connection Mode: DirectTcp

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.

To import to the DocumentDB API (Sequential Record Import)

The Azure Cosmos DB 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) Azure Cosmos DB collection, as well as sharded import whereby data is partitioned across multiple single-partition and/or multi-partition Azure Cosmos DB collections. For more information about partitioning data, see Partitioning and scaling in Azure Cosmos DB.



DocumentDB Data Migration Tool

Target Information

Specify target information

Export to: DocumentDB - Sequential record import (partitioned collection)

Connection String: cumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;

Collection: employees[1-4]

Partition Key: fullName

Collection Throughput: 1000

Id Field: empid

The format of the Azure Cosmos DB connection string is:

```
AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;
```

The Azure Cosmos DB account connection string can be retrieved from the Keys blade of the Azure portal, as described in How to manage an Azure Cosmos DB account, however the name of the database needs to be appended to the connection string in the following format:

```
Database=<Azure Cosmos DB Database>;
```

> NOTE
>
> Use the Verify command to ensure that the Azure Cosmos DB 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

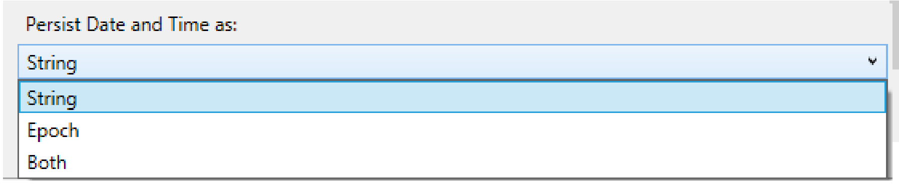
250,000 RUs). For best import performance, choose a higher throughput. For more information about performance levels, see Performance levels in Azure Cosmos DB. Any import to collections with throughput >10,000 RUs will require a partition key. If you choose to have more than 250,000 RUs, you will need to file a request in the portal to have your account increased.

> 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 Azure Cosmos DB 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).

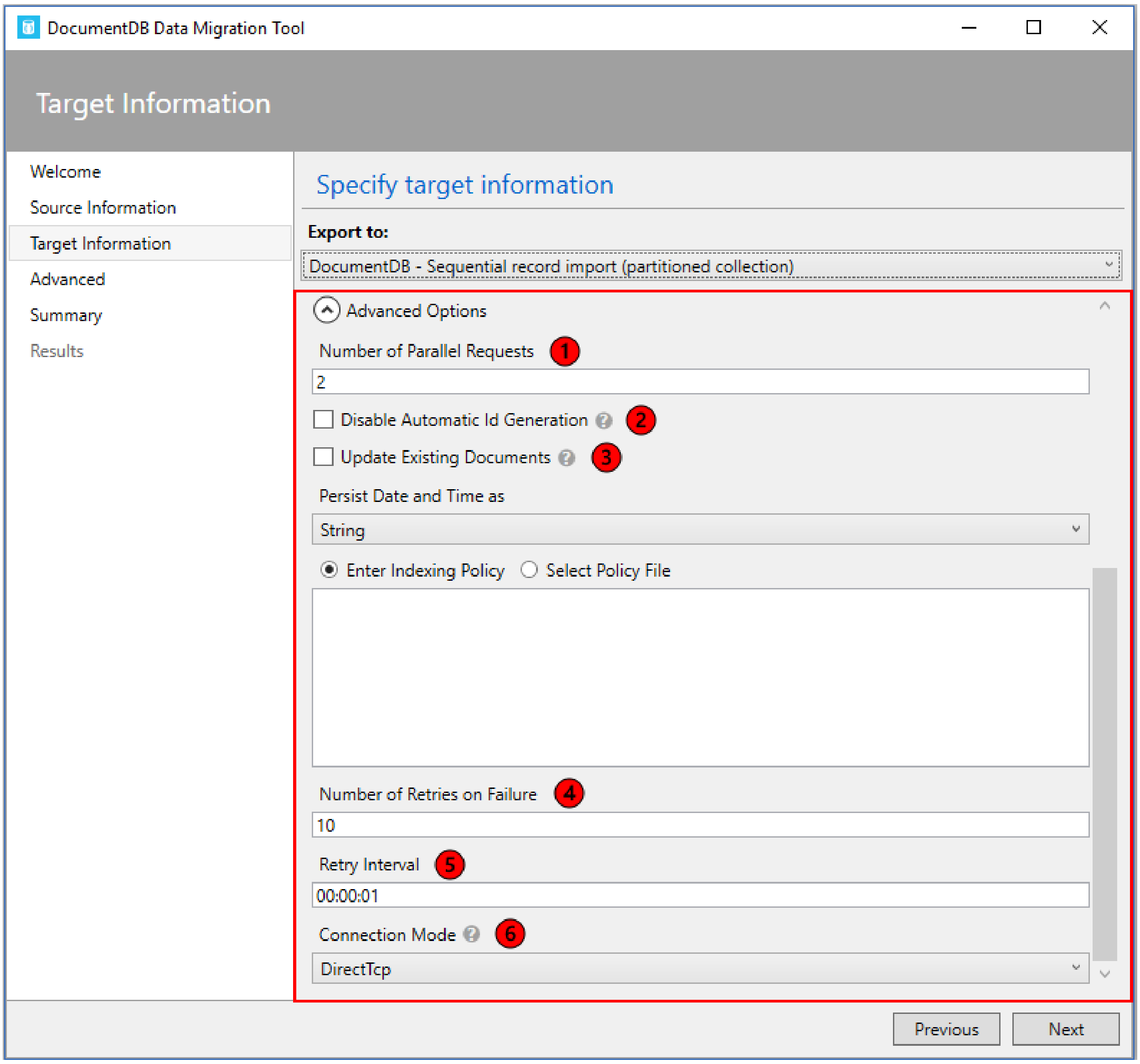
There are a number of advanced options available during import. First, when importing date types (e.g. from SQL Server or MongoDB), you can choose between three import options:


Persist Date and Time as: String

- 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 Azure Cosmos DB - 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 Azure Cosmos DB in case of transient failures (e.g. network connectivity interruption).
5. Retry Interval: Specifies how long to wait between retrying the connection to Azure Cosmos DB in case of transient failures (e.g. network connectivity interruption).
6. Connection Mode: Specifies the connection mode to use with Azure Cosmos DB. 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

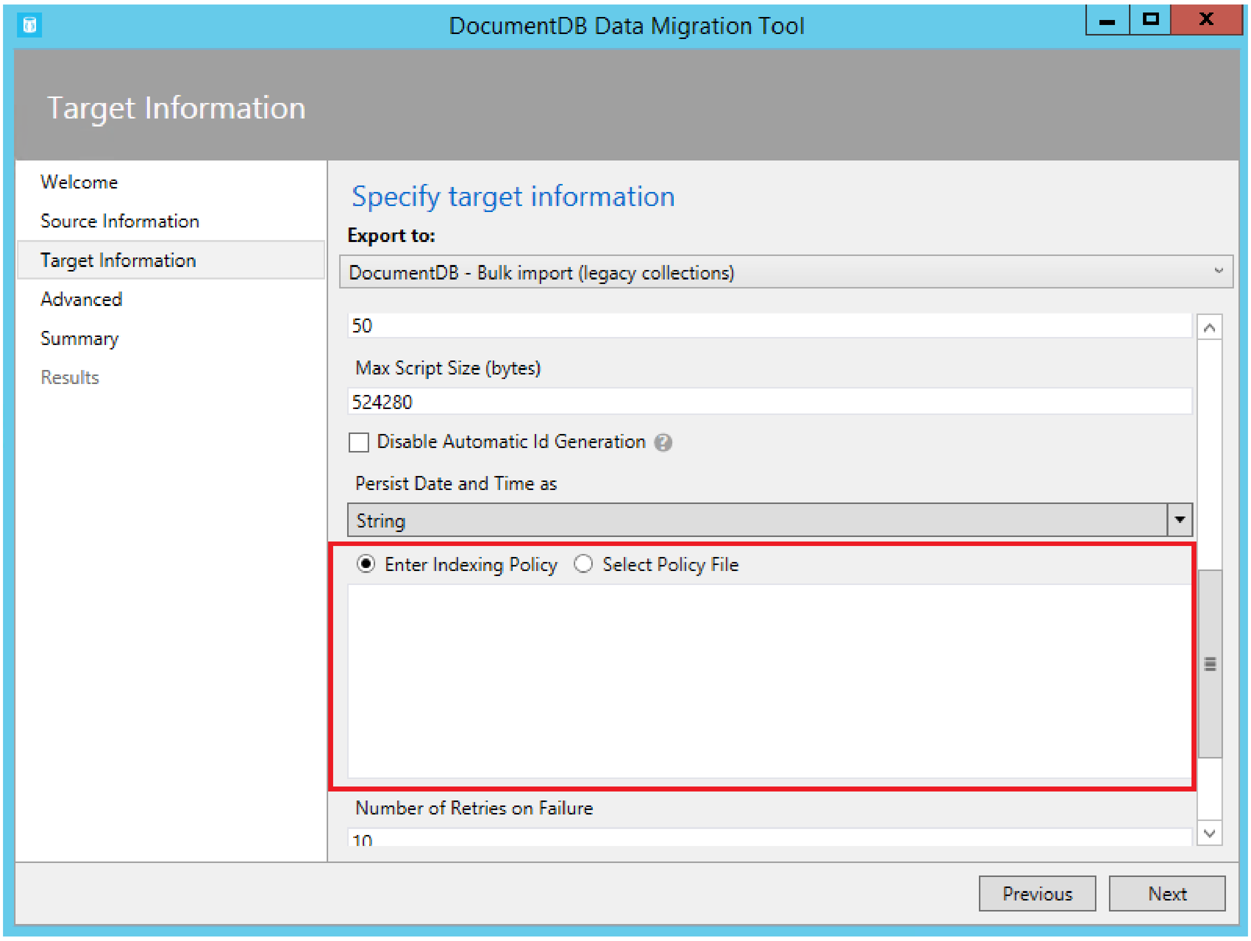
Connection Mode (6): DirectTcp

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 Azure Cosmos DB 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 Azure Cosmos DB Bulk import and Azure Cosmos DB Sequential record options, navigate to the Indexing Policy section.


DocumentDB Data Migration Tool

Target Information

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

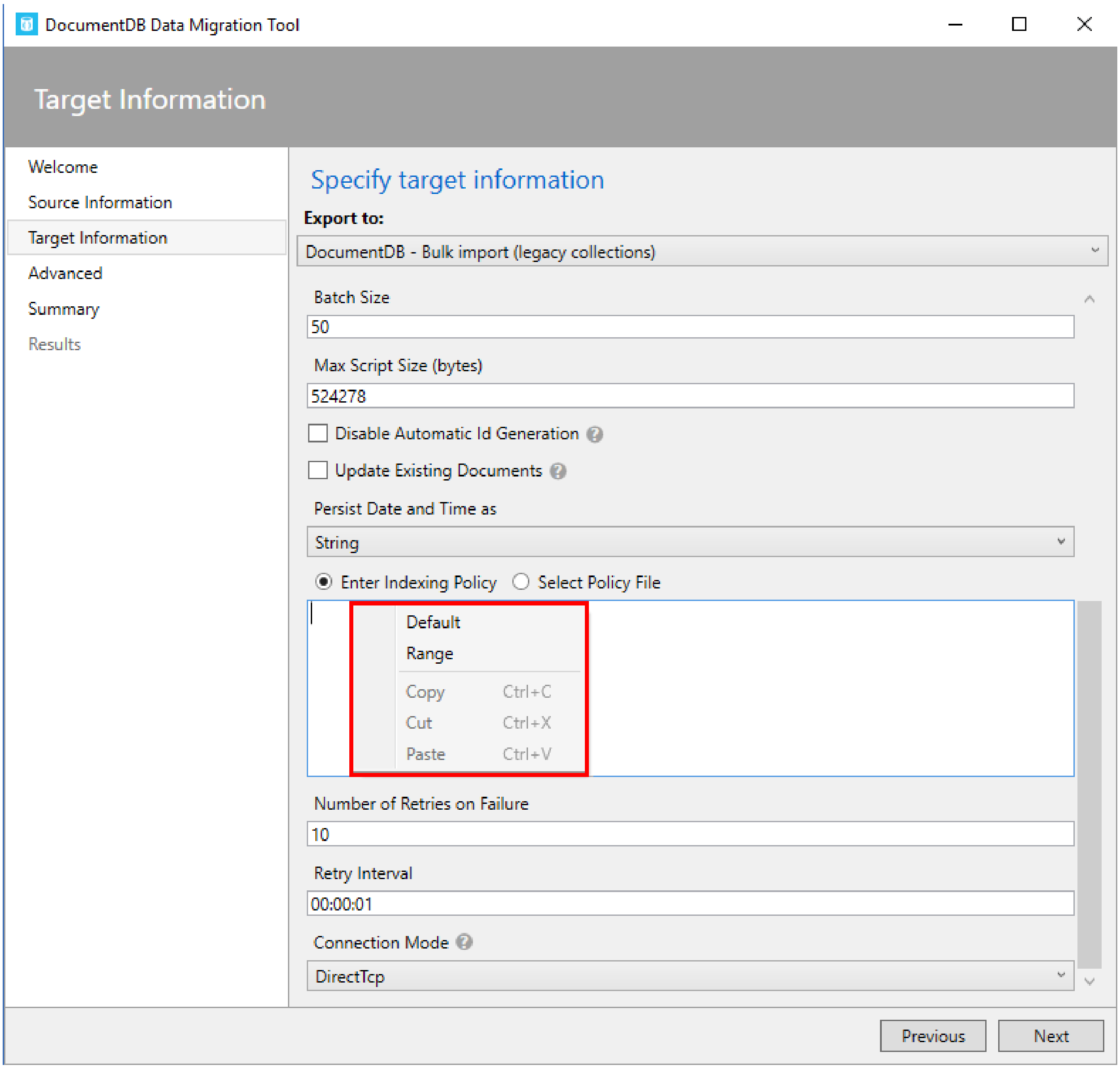
● Enter Indexing Policy ○ Select Policy File

Number of Retries on Failure: 10

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.



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

Number of Retries on Failure: 10

Retry Interval: 00:00:01

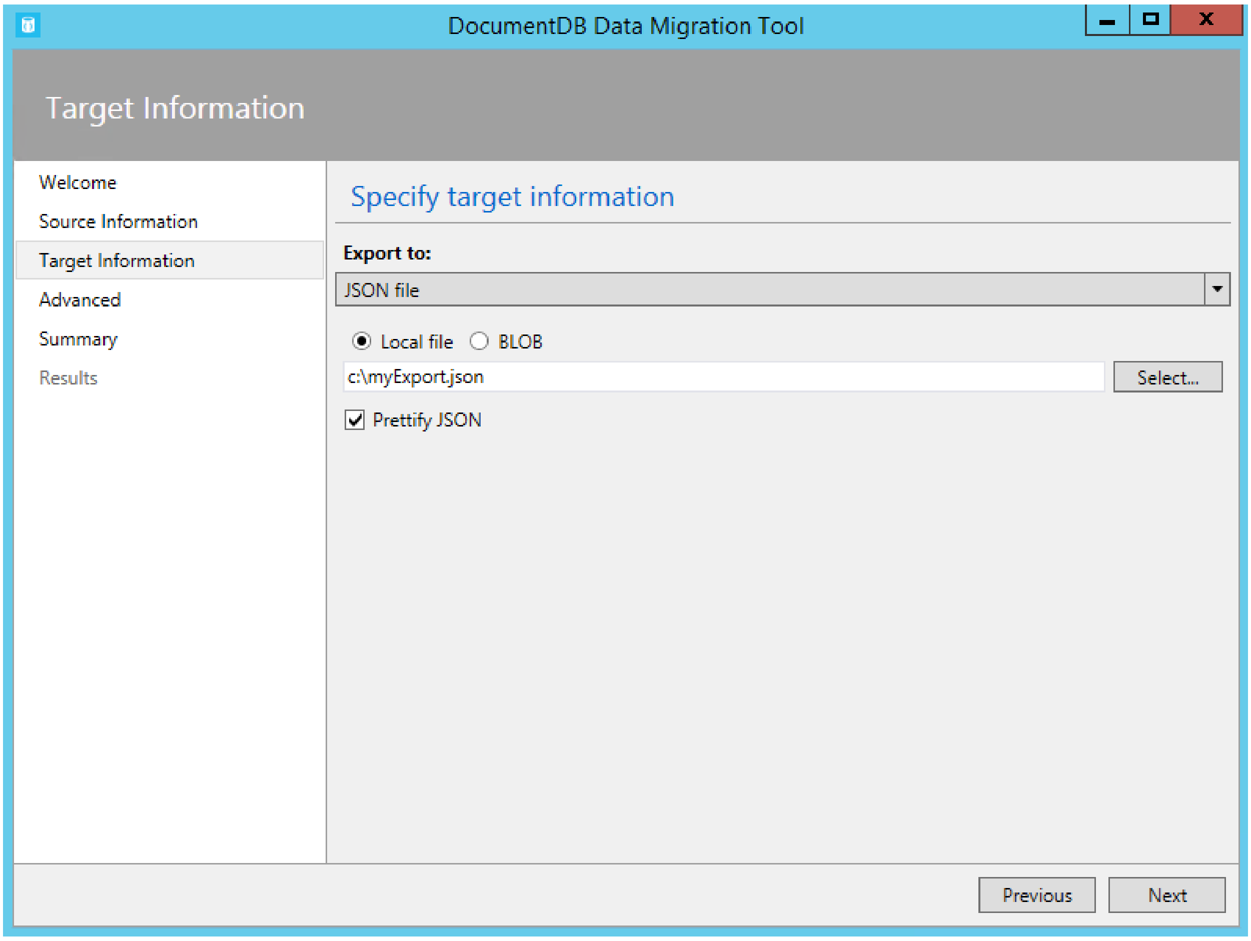
Connection Mode: DirectTcp

NOTE

If you do not specify an indexing policy, then the default policy will be applied. For more information about indexing policies, see Azure Cosmos DB indexing policies.

Export to JSON file

The Azure Cosmos DB 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.



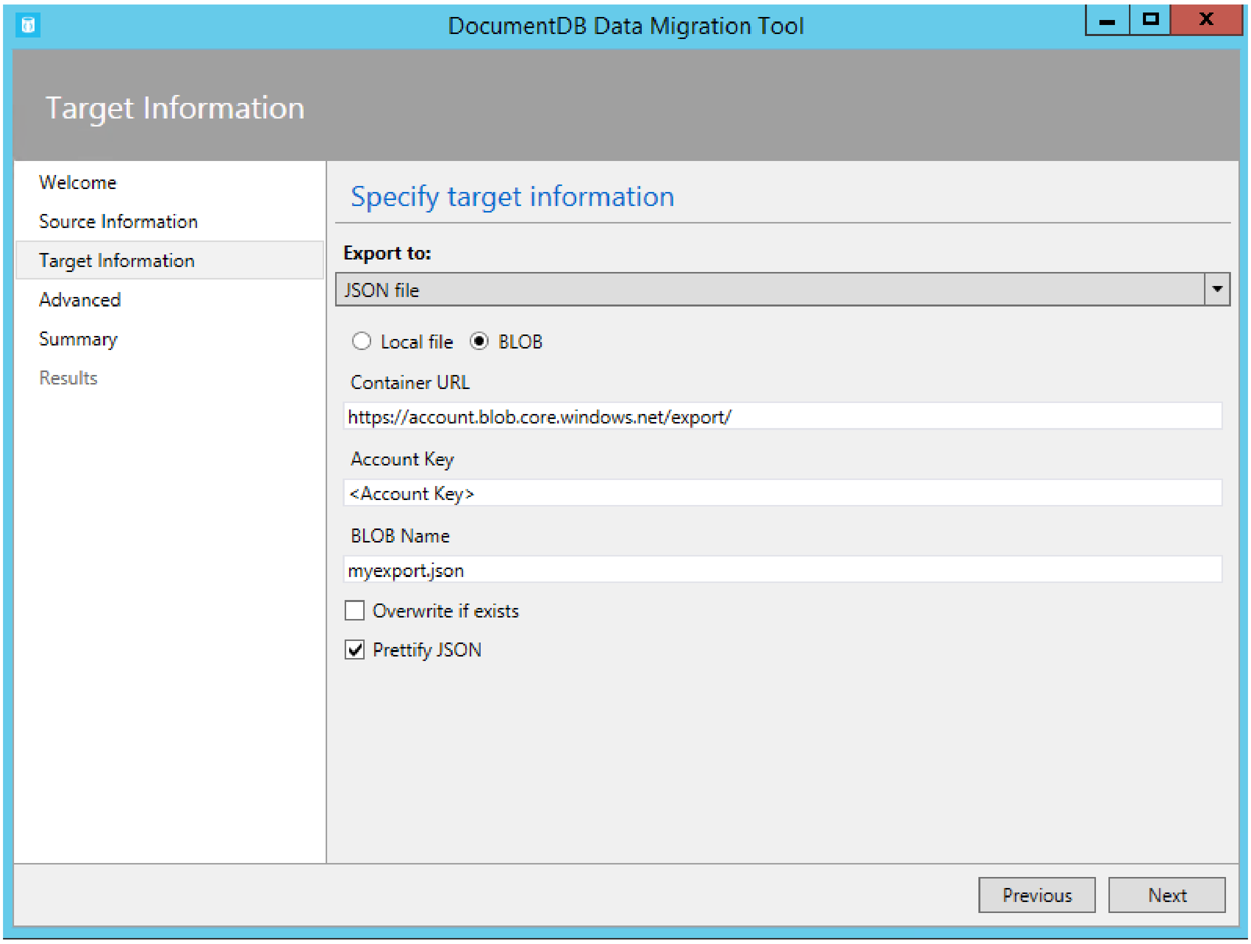
Specify target information

Export to: JSON file

● Local file ○ BLOB

c:\myExport.json

☑ Prettify JSON



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

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 Azure Cosmos DB 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 Azure Cosmos DB 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:

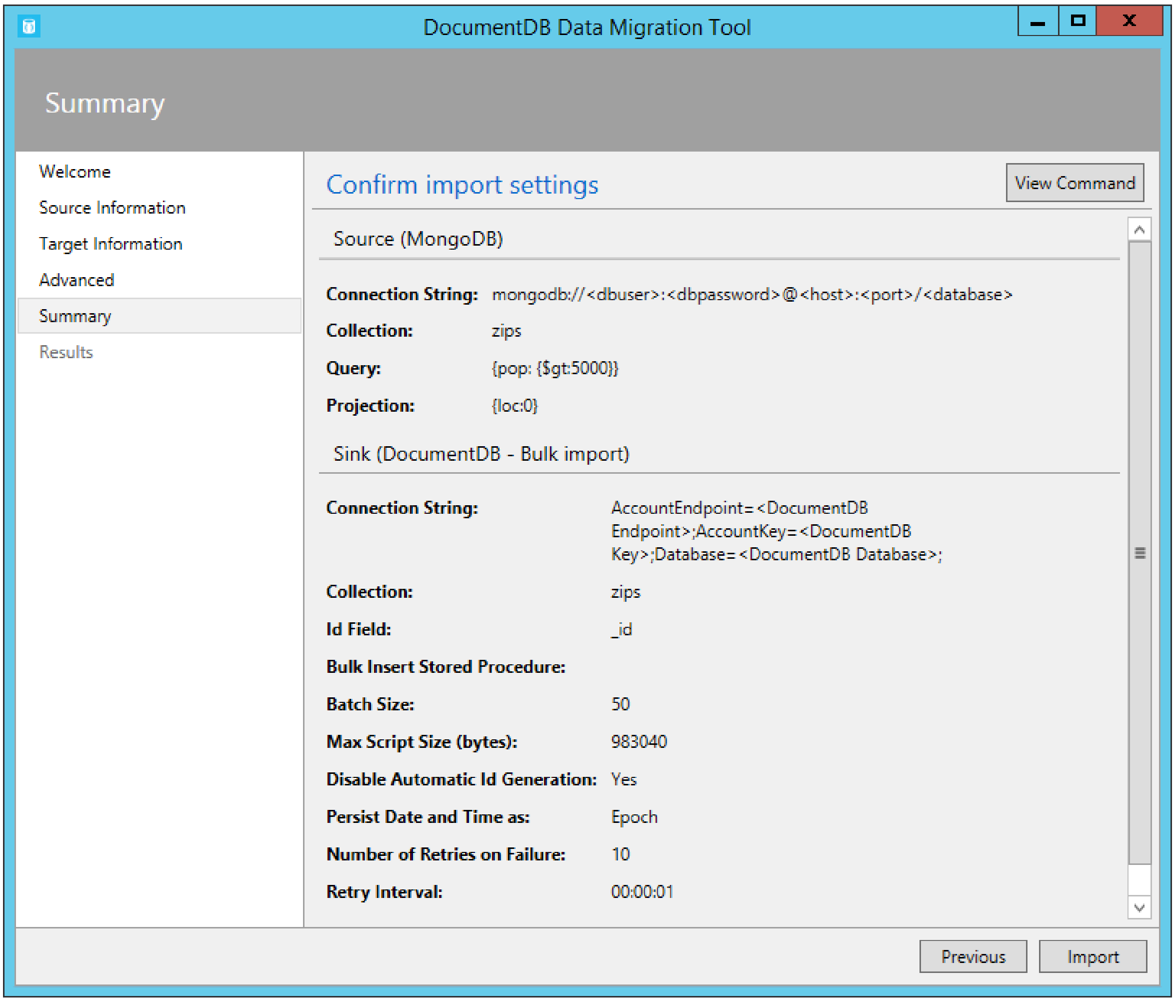
1. If a file name is not provided, then all errors will be returned on the Results page.
2. If a file name is provided without a directory, then the file will be created (or overwritten) in the current environment directory.
3. If you select an existing file, then the file will be overwritten, there 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/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):



Confirm import settings

Source (MongoDB)

Connection String: mongodb://<dbuser>:<dbpassword>@<host>:<port>/<database>

Collection: zips

Query: {pop: {$gt:5000}}

Projection: {loc:0}

Sink (DocumentDB - Bulk import)

Connection String: AccountEndpoint=<DocumentDB Endpoint>;AccountKey=<DocumentDB Key>;Database=<DocumentDB Database>;

Collection: zips

Id Field: _id

Bulk Insert Stored Procedure:

Batch Size: 50

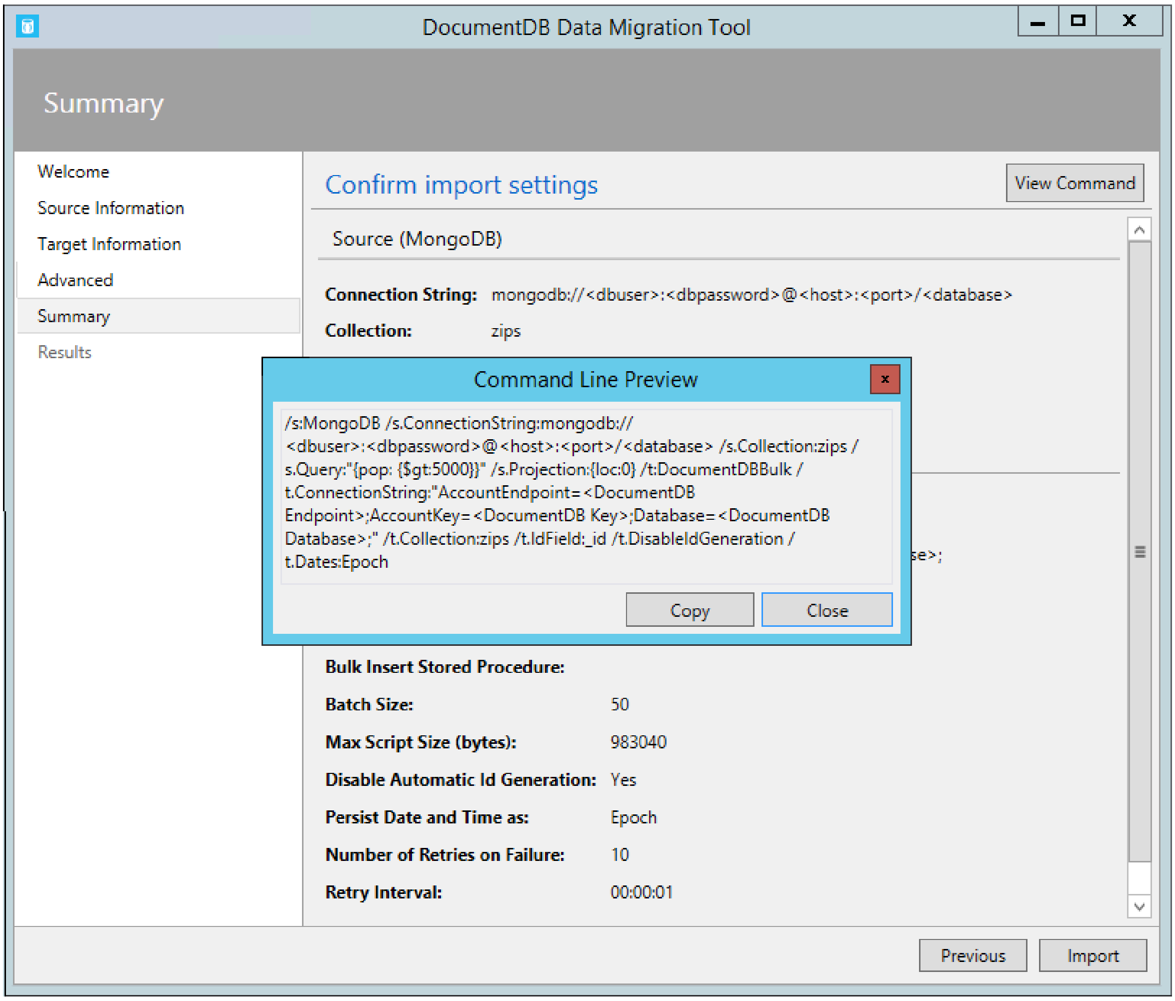
Max Script Size (bytes): 983040

Disable Automatic Id Generation: Yes

Persist Date and Time as: Epoch

Number of Retries on Failure: 10

Retry Interval: 00:00:01



Confirm import settings

Source (MongoDB)

Connection String: mongodb://<dbuser>:<dbpassword>@<host>:<port>/<database>

Collection: zips

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
```

Bulk Insert Stored Procedure:

Batch Size: 50

Max Script Size (bytes): 983040

Disable Automatic Id Generation: Yes

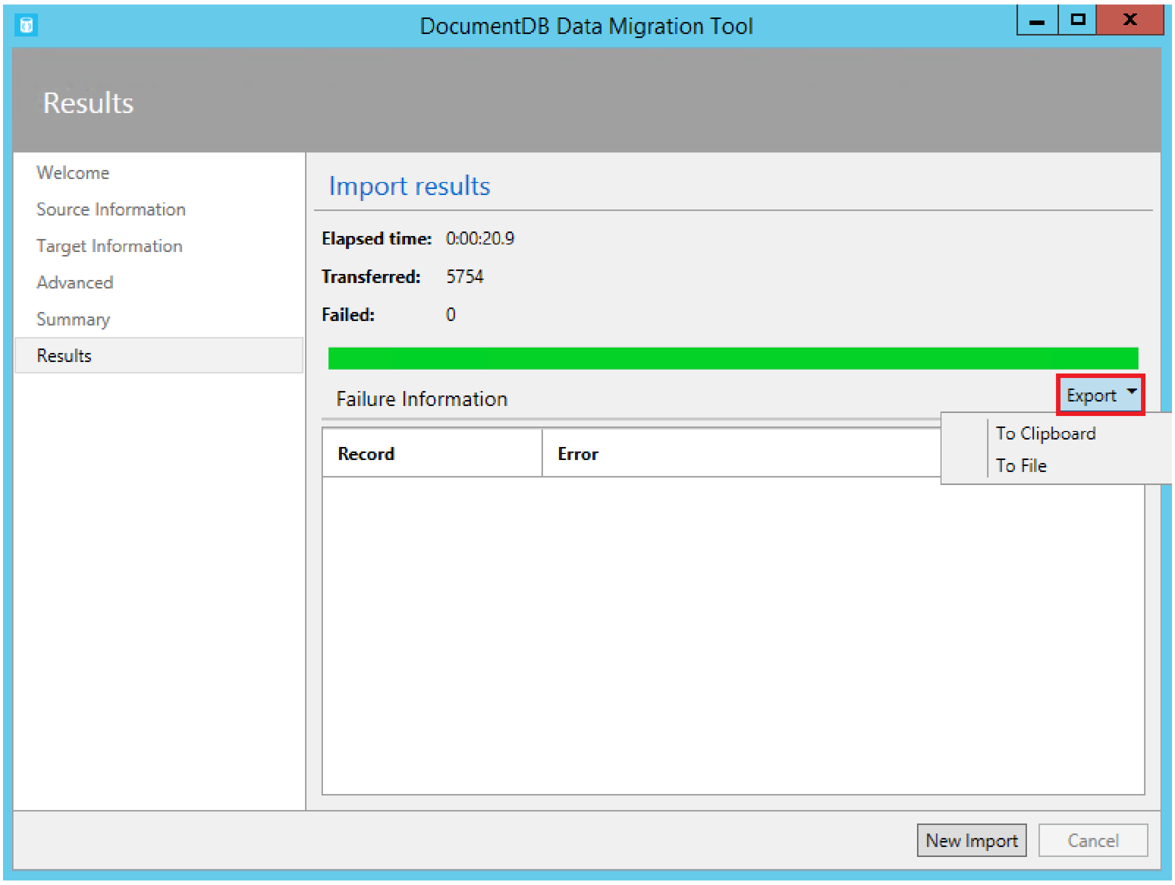
Persist Date and Time as: Epoch

Number of Retries on Failure: 10

Retry Interval: 00:00:01

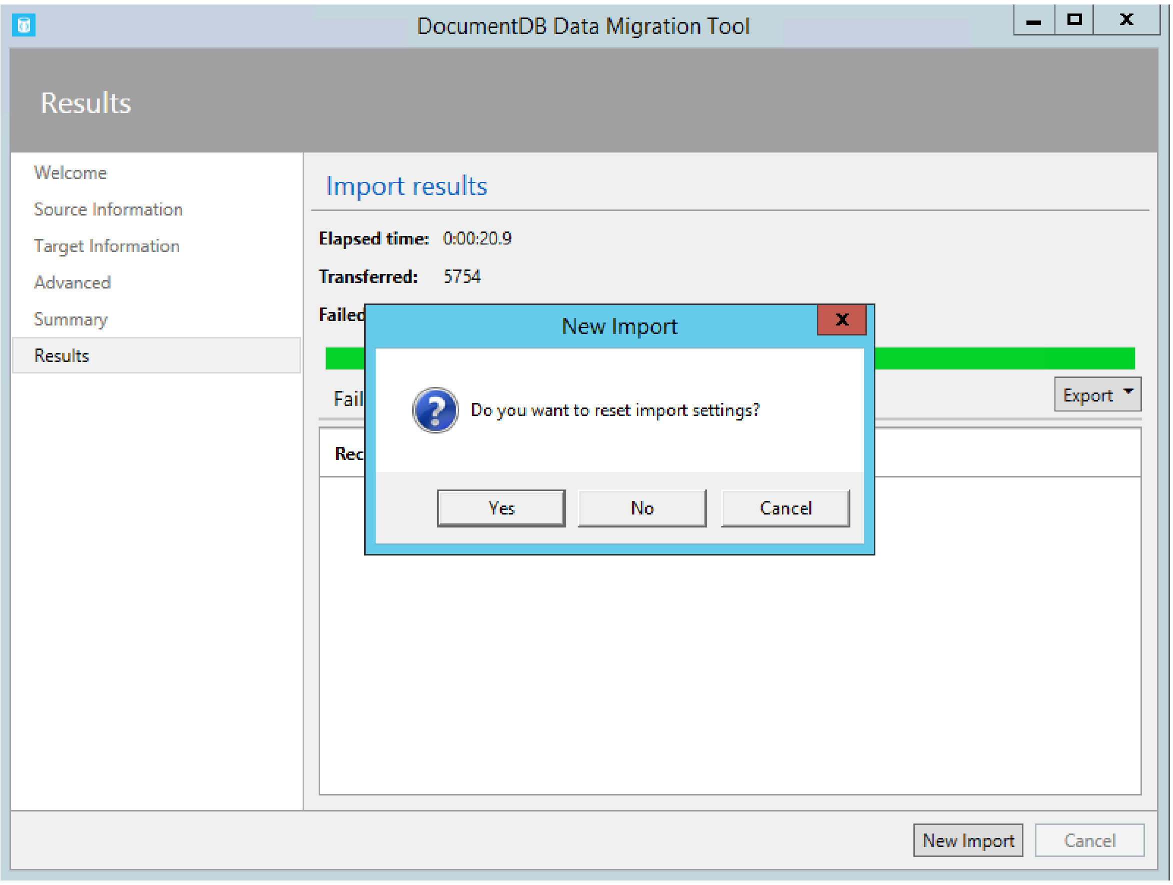
2. Once you're satisfied with your source and target options, click **Import**. The elapsed time, transferred

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).



Elapsed time: 0:00:20.9
Transferred: 5754
Failed: 0

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.



Elapsed time: 0:00:20.9
Transferred: 5754

New Import

Do you want to reset import settings?

Next steps

In this tutorial, you've done the following:

- Installed the Data Migration tool
- Imported data from different data sources
- Exported from Azure Cosmos DB to JSON

You can now proceed to the next tutorial and learn how to query data using Azure Cosmos DB.

How to query data?

Azure Cosmos DB: How to import MongoDB data?

6/13/2017 • 4 min to read • Edit Online

To migrate data from MongoDB to an Azure Cosmos DB account for use with the MongoDB API, you must:

- Download either *mongoimport.exe* or *mongorestore.exe* from the MongoDB Download Center.
- Get your API for MongoDB connection string.

If you are importing data from MongoDB and plan to use it with the DocumentDB API, you should use the Data Migration Tool to import data. For more information, see Data Migration Tool.

This tutorial covers the following tasks:

- Retrieving your connection string
- Importing MongoDB data using mongoimport
- Importing MongoDB data using mongorestore

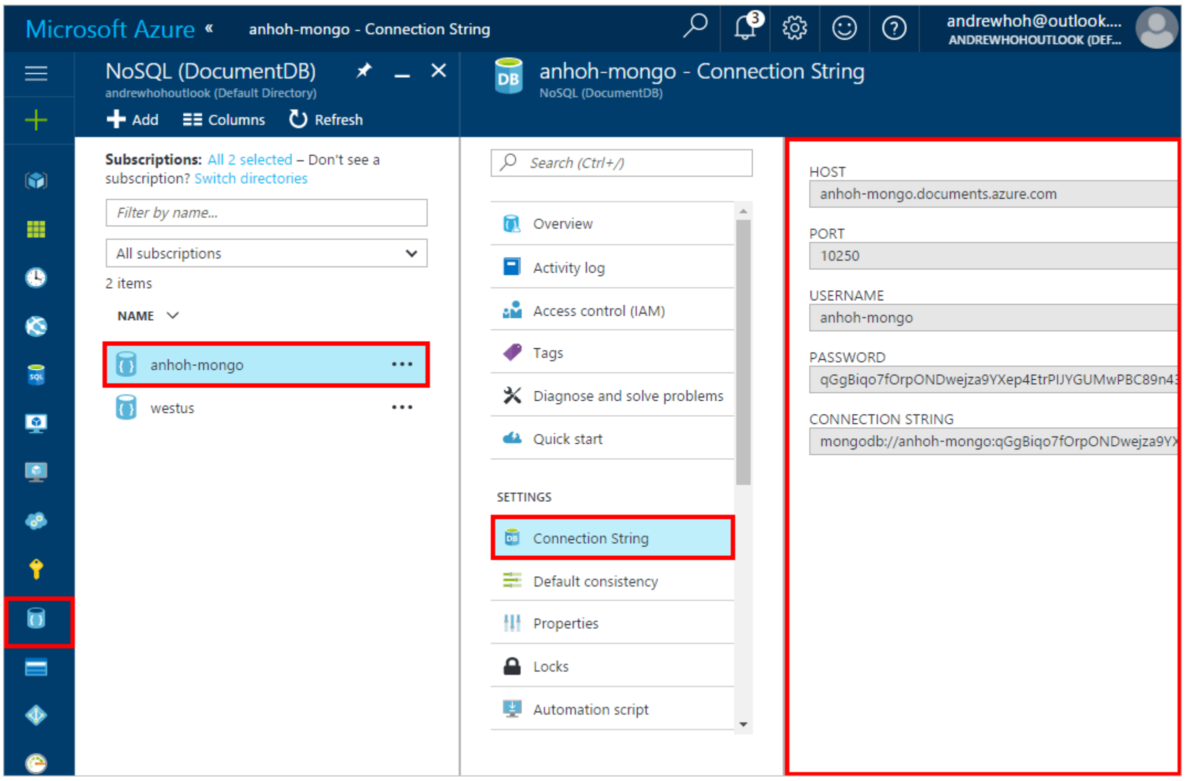
Prerequisites

- Increase throughput: The duration of your data migration depends on the amount of throughput you set up for your collections. Be sure to increase the throughput for larger data migrations. After you've completed the migration, decrease the throughput to save costs. For more information about increasing throughput in the Azure portal, see Performance levels and pricing tiers in Azure Cosmos DB.

- Enable SSL: Azure Cosmos DB has strict security requirements and standards. Be sure to enable SSL when you interact with your account. The procedures in the rest of the article include how to enable SSL for *mongoimport* and *mongorestore*.

Find your connection string information (host, port, username, and password)

1. In the Azure portal, in the left pane, click the **Azure Cosmos DB** entry.
2. In the **Subscriptions** pane, select your account name.
3. In the **Connection String** blade, click **Connection String**.
   The right pane contains all the information you need to successfully connect to your account.



HOST
anhoh-mongo.documents.azure.com

PORT
10250

USERNAME
anhoh-mongo

PASSWORD
qGgBiqo7fOrpONDwejza9YXep4EtrPlJYGUMwPBC89n43

CONNECTION STRING
mongodb://anhoh-mongo:qGgBiqo7fOrpONDwejza9YX

Import data to MongoDB API with mongoimport

To import data to your Azure Cosmos DB account, use the following template to execute the import. Fill in *host*, *username*, and *password* with the values that are specific to your account.

Template:

```
mongoimport.exe --host <your_hostname>:10255 -u <your_username> -p <your_password> --db <your_database> --collection
<your_collection> --ssl --sslAllowInvalidCertificates --type json --file C:\sample.json
```

Example:

```
mongoimport.exe --host anhoh-host.documents.azure.com:10255 -u anhoh-host -p
tkvaVkp4Nnaoirnouenrgisuner2435qwefBH0z256Na24frio34LNQasfaefarfernoimczciqisAXw== --ssl --sslAllowInvalidCertificates --db
sampleDB --collection sampleColl --type json --file C:\Users\anhoh\Desktop\*.json
```

Import data to API for MongoDB with mongorestore

To restore data to your API for MongoDB account, use the following template to execute the import. Fill in *host*, *username*, and *password* with the values specific to your account.

Template:

```
mongorestore.exe --host <your_hostname>:10255 -u <your_username> -p <your_password> --db <your_database> --collection
<your_collection> --ssl --sslAllowInvalidCertificates <path_to_backup>
```

Example:

```
mongorestore.exe --host anhoh-host.documents.azure.com:10255 -u anhoh-host -p
tkvaVkp4Nnaoirnouenrgisuner2435qwefBH0z256Na24frio34LNQasfaefarfernoimczciqisAXw== --ssl --sslAllowInvalidCertificates
./dumps/dump-2016-12-07
```

Guide for a successful migration

1. Pre-create and scale your collections

   - By default, Azure Cosmos DB will provision a new MongoDB collection with 1,000 RUs. Before the migration using mongoimport, mongorestore, or mongomirror, pre-create all your collections from the Azure Portal or MongoDB drivers, tools, etc. If your collection is greater than 10GB, make sure to create a sharded / partitioned collection with an appropriate shard key.

   - From the Azure Portal, increase your collections' throughput from 1,000 RUs for a single partition collection / 2,500 RUs for a sharded collection just for the migration. With the higher throughput you will be able to avoid throttling and migrate in a shorter period of time. With Azure Cosmos DB's hourly billing, you can reduce the throughput immediately after the migration to save costs.

2. Calculate the approximate RU charge for a single document write

   - Connect to your Azure Cosmos DB MongoDB database from the MongoDB Shell. Instructions can be found here.

   - Run a sample insert command using one of your sample documents from the MongoDB Shell

     `db.coll.insert({ "playerId": "a067ff", "hashedid": "bb0091", "countryCode": "hk" })`

   - Following the insert, run `db.runCommand({getLastRequestStatistics: 1})` and you will receive a response as such

     ```
     globaldb:PRIMARY> db.runCommand({getLastRequestStatistics: 1})
     {
        "_t": "GetRequestStatisticsResponse",
        "ok": 1,
        "CommandName": "insert",
        "RequestCharge": 10,
        "RequestDurationInMilliSeconds": NumberLong(50)
     }
     ```

   - Take note of the Request Charge

3. Determine the latency from your machine to the Azure Cosmos DB cloud service.

   - Enable verbose logging from the MongoDB Shell with the command: `setVerboseShell(true)`

   - Run a simple query against the database: `db.coll.find().limit(1)` and you will receive a response as such

     ```
     Fetched 1 record(s) in 100(ms)
     ```

4. Make sure to remove the inserted document before the migration to ensure no duplicate documents. You can remove the documents with a `db.coll.remove({})`.

5. Calculating the approximate *batchSize* and *numInsertionWorkers*

   - For the *batchSize*, divide the total provisioned RUs by the RUs consumed from your single document write in Step 3.

   - If the calculated *batchSize* <= 24, you use that number as your *batchSize*

   - If the calculated *batchSize* > 24, you should set the *batchSize* to 24.

   - For the *numInsertionWorkers*, use this equation: *numInsertionWorkers = (provisioned throughput * latency in seconds) / (batch size * consumed RUs for a single write)*

PROPERTY	VALUE
batchSize	24
RUs provisioned	10000
Latency	0.100 s
RU charged for 1 doc write	10 RUs
numInsertionWorkers	?

numInsertionWorkers = (10000RUs x 0.1s) / (24 x 10 RUs) = 4.1666

6. Final migration command:

```
mongoimport.exe --host anhoh-mongodb.documents.azure.com:10255 -u anhoh-mongodb -p
wzRJCyjtLPNuhm53yTwaefawuiefhbauwebhfuabweifbiauweb2YVdl2ZFNZNv8IU89LqFVm5U0bw== --ssl --sslAllowInvalidCertificates --
jsonArray --db dabasename --collection collectionName --file "C:\sample.json" --numInsertionWorkers 4 --batchSize 24
```

Next steps

In this tutorial, you've done the following:

- Retrieved your connection string
- Imported MongoDB data using mongoimport
- Imported MongoDB data using mongorestore

You can now proceed to the next tutorial and learn how to query MongoDB data using Azure Cosmos DB.

How to query MongoDB data?

Azure Cosmos DB: How to query using SQL?

5/31/2017 • 1 min to read • Edit Online

The Azure Cosmos DB DocumentDB API supports querying documents using SQL. This article provides a sample document and two sample SQL queries and results.

This article covers the following tasks:

- Querying data with SQL

Sample document

The SQL queries in this article use the following sample 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
}
```

Where can I run SQL queries?

You can run queries using the Data Explorer in the Azure portal, via the REST API and SDKs, and even the Query playground, which runs queries on an existing set of sample data.

For more information about SQL queries, see:

- SQL query and SQL syntax

Prerequisites

This tutorial assumes you have an Azure Cosmos DB account and collection. Don't have any of those? Complete the 5-minute quickstart or the developer tutorial to create an account and collection.

Example query 1

Given the sample family document above, following SQL query returns the documents where the id field matches `WakefieldFamily`. Since it's a `SELECT *` statement, the output of the query is the complete JSON document:

Query

```
SELECT *
FROM Families f
WHERE f.id = "WakefieldFamily"
```

Results

```
{
 "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
}
```

Example query 2

The next query returns all the given names of children in the family whose id matches `WakefieldFamily` ordered by their grade.

Query

```
SELECT c.givenName
FROM Families f
JOIN c IN f.children
WHERE f.id = 'WakefieldFamily'
ORDER BY f.children.grade ASC
```

Results

```
[
  { "givenName": "Jesse" },
  { "givenName": "Lisa"}
]
```

Next steps

In this tutorial, you've done the following:

- Learned how to query using SQL

You can now proceed to the next tutorial to learn how to distribute your data globally.

Distribute your data globally

Azure Cosmos DB: How to query with API for MongoDB?

5/30/2017 • 3 min to read • Edit Online

The Azure Cosmos DB API for MongoDB supports MongoDB shell queries.

This article covers the following tasks:

- Querying data with MongoDB

Sample document

The queries in this article use the following sample 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
}
```

Example query 1

Given the sample family document above, the following query returns the documents where the id field matches `WakefieldFamily`.

Query

```
db.families.find({ id: “WakefieldFamily”})
```

Results

```
{
"_id": "ObjectId(\"58f65e1198f3a12c7090e68c\")",
"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
}
```

Example query 2

The next query returns all the children in the family.

Query

```
db.familes.find( { id: “WakefieldFamily” }, { children: true } )
```

Results

```
{
"_id": "ObjectId("58f65e1198f3a12c7090e68c")",
"children": [
  {
    "familyName": "Merriam",
    "givenName": "Jesse",
    "gender": "female",
    "grade": 1,
    "pets": [
      { "givenName": "Goofy" },
      { "givenName": "Shadow" }
    ]
  },
  {
    "familyName": "Miller",
    "givenName": "Lisa",
    "gender": "female",
    "grade": 8
  }
]
}
```

Example query 3

The next query returns all the families which are registered.

Query

```
db.families.find( { "isRegistered" : true })
```

Results No document will be returned.

Example query 4

The next query returns all the families which are not registered.

Query

```
db.families.find( { "isRegistered" : false })
```

Results

```
 {
"_id": ObjectId("58f65e1198f3a12c7090e68c"),
"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
```

}

Example query 5

The next query returns all the families which are not registered and state is NY.

Query

```
db.families.find( { "isRegistered" : false, "address.state" : "NY" })
```

Results

```
 {
"_id": ObjectId("58f65e1198f3a12c7090e68c"),
"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
```

}

Example query 6

The next query returns all the families where children grades are 8.

Query

```
db.families.find( { children : { $elemMatch: { grade : 8 }} } )
```

Results

```
 {
"_id": ObjectId("58f65e1198f3a12c7090e68c"),
"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
```

}

Example query 7

The next query returns all the families where size of children array is 3.

Query

```
db.Family.find( {children: { $size:3} } )
```

Results

No results will be returned as we do not have more than 2 children. Only when parameter is 2 this query will succeed and return the full document.

Next steps

In this tutorial, you've done the following:

- Learned how to query using MongoDB

You can now proceed to the next tutorial to learn how to distribute your data globally.

Distribute your data globally

Azure Cosmos DB: How to query table data by using the Table API (preview)?

5/30/2017 • 2 min to read • Edit Online

The Azure Cosmos DB Table API (preview) supports OData and LINQ queries against key/value (table) data.

This article covers the following tasks:

- Querying data with the Table API

The queries in this article use the following sample `People` table:

PARTITIONKEY	ROWKEY	EMAIL	PHONENUMBER
Harp	Walter	Walter@contoso.com	425-555-0101
Smith	Ben	Ben@contoso.com	425-555-0102
Smith	Jeff	Jeff@contoso.com	425-555-0104

Because Azure Cosmos DB is compatible with the Azure Table storage APIs, see Querying Tables and Entities for details on how to query by using the Table API.

For more information on the premium capabilities that Azure Cosmos DB offers, see Azure Cosmos DB: Table API and Develop with the Table API in .NET.

Prerequisites

For these queries to work, you must have an Azure Cosmos DB account and have entity data in the container. Don't have any of those? Complete the five-minute quickstart or the developer tutorial to create an account and populate your database.

Query on PartitionKey and RowKey

Because the PartitionKey and RowKey properties form an entity's primary key, you can use the following special syntax to identify the entity:

Query

```
https://<mytableendpoint>/People(PartitionKey='Harp',RowKey='Walter')
```

Results

PARTITIONKEY	ROWKEY	EMAIL	PHONENUMBER
Harp	Walter	Walter@contoso.com	425-555-0104

Alternatively, you can specify these properties as part of the `$filter` option, as shown in the following section. Note that the key property names and constant values are case-sensitive. Both the PartitionKey and RowKey properties are of type String.

Query by using an OData filter

When you're constructing a filter string, keep these rules in mind:

- Use the logical operators defined by the OData Protocol Specification to compare a property to a value. Note that you can't compare a property to a dynamic value. One side of the expression must be a constant.
- The property name, operator, and constant value must be separated by URL-encoded spaces. A space is URL-encoded as `%20`.
- All parts of the filter string are case-sensitive.
- The constant value must be of the same data type as the property in order for the filter to return valid results. For more information about supported property types, see Understanding the Table Service Data Model.

Here's an example query that shows how to filter by the PartitionKey and Email properties by using an OData `$filter`.

Query

```
https://<mytableapi-endpoint>/People()?$filter=PartitionKey%20eq%20'Smith'%20and%20Email%20eq%20'Ben@contoso.com'
```

For more information on how to construct filter expressions for various data types, see Querying Tables and Entities.

Results

PARTITIONKEY	ROWKEY	EMAIL	PHONENUMBER
Ben	Smith	Ben@contoso.com	425-555-0102

Query by using LINQ

You can also query by using LINQ, which translates to the corresponding OData query expressions. Here's an example of how to build queries by using the .NET SDK:

```
CloudTableClient tableClient = account.CreateCloudTableClient();
CloudTable table = tableClient.GetTableReference("people");

TableQuery<CustomerEntity> query = new TableQuery<CustomerEntity>()
    .Where(
        TableQuery.CombineFilters(
            TableQuery.GenerateFilterCondition(PartitionKey, QueryComparisons.Equal, "Smith"),
            TableOperators.And,
            TableQuery.GenerateFilterCondition(Email, QueryComparisons.Equal,"Ben@contoso.com")
    ));

await table.ExecuteQuerySegmentedAsync<CustomerEntity>(query, null);
```

Next steps

In this tutorial, you've done the following:

- Learned how to query by using the Table API (preview)

You can now proceed to the next tutorial to learn how to distribute your data globally.

Distribute your data globally

Azure Cosmos DB: How to query with the Graph API (preview)?

5/30/2017 • 1 min to read • Edit Online

The Azure Cosmos DB Graph API (preview) supports Gremlin queries. This article provides sample documents and queries to get you started. A detailed Gremlin reference is provided in the Gremlin support article.

This article covers the following tasks:

- Querying data with Gremlin

Prerequisites

For these queries to work, you must have an Azure Cosmos DB account and have graph data in the container. Don't have any of those? Complete the 5-minute quickstart or the developer tutorial to create an account and populate your database. You can run the following queries using the Azure Cosmos DB .NET graph library, Gremlin console, or your favorite Gremlin driver.

Count vertices in the graph

The following snippet shows how to count the number of vertices in the graph:

```
g.V().count()
```

Filters

You can perform filters using Gremlin's `has` and `hasLabel` steps, and combine them using `and`, `or`, and `not` to build more complex filters. Azure Cosmos DB provides schema-agnostic indexing of all properties within your vertices and degrees for fast queries:

```
g.V().hasLabel('person').has('age', gt(40))
```

Projection

You can project certain properties in the query results using the `values` step:

```
g.V().hasLabel('person').values('firstName')
```

Find related edges and vertices

So far, we've only seen query operators that work in any database. Graphs are fast and efficient for traversal operations when you need to navigate to related edges and vertices. Let's find all friends of Thomas. We do this by using Gremlin's `outE` step to find all the out-edges from Thomas, then traversing to the in-vertices from those edges using Gremlin's `inV` step:

```
g.V('thomas').outE('knows').inV().hasLabel('person')
```

The next query performs two hops to find all of Thomas' "friends of friends", by calling `outE` and `inV` two times.

```
g.V('thomas').outE('knows').inV().hasLabel('person').outE('knows').inV().hasLabel('person')
```

You can build more complex queries and implement powerful graph traversal logic using Gremlin, including mixing filter expressions, performing looping using the `loop` step, and implementing conditional navigation using the `choose` step. Learn more about what you can do with Gremlin support!

Next steps

In this tutorial, you've done the following:

- Learned how to query using Graph

You can now proceed to the next tutorial to learn how to distribute your data globally.

Distribute your data globally

How to setup Azure Cosmos DB global distribution using the DocumentDB API

5/30/2017 • 7 min to read • Edit Online

In this article, we show how to use the Azure portal to setup Azure Cosmos DB global distribution and then connect using the DocumentDB API.

This article covers the following tasks:

- Configure global distribution using the Azure portal
- Configure global distribution using the DocumentDB APIs

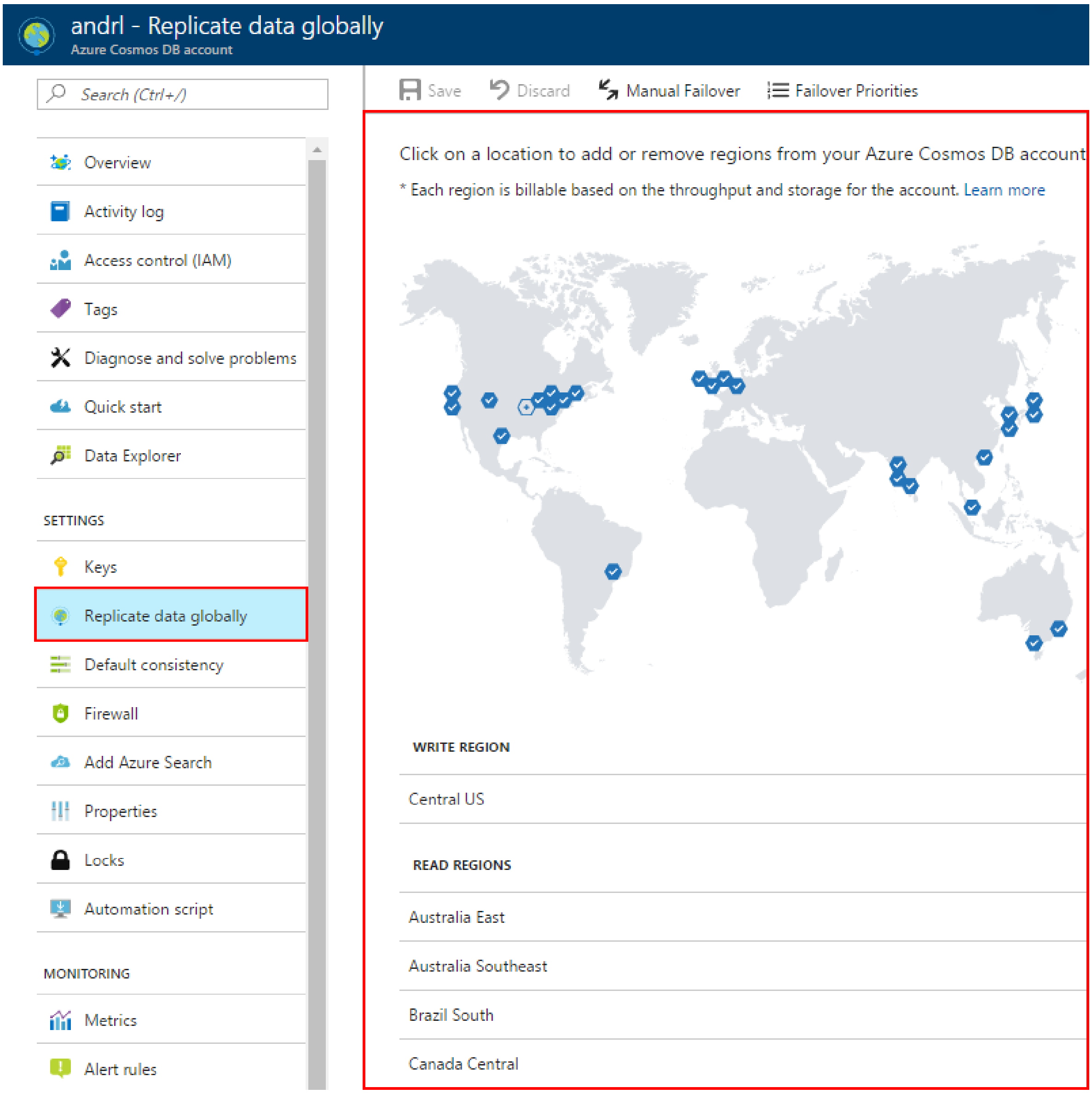
You can learn about Azure Cosmos DB global distribution in this Azure Friday video with Scott Hanselman and Principal Engineering Manager Karthik Raman.

For more information about how global database replication works in Cosmos DB, see Distribute data globally with Cosmos DB.

Add global database regions using the Azure Portal

Azure Cosmos DB is available in all Azure regions world-wide. 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 left bar, click **Azure Cosmos DB**.
2. In the **Azure Cosmos DB** 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 by clicking regions in the map, 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.



Click on a location to add or remove regions from your Azure Cosmos DB account

* Each region is billable based on the throughput and storage for the account. Learn more

WRITE REGION

Central US

READ REGIONS

Australia East

Australia Southeast

Brazil South

Canada Central

Once you add a second region, the **Manual Failover** option is enabled on the **Replicate data globally** blade in the portal. You can use this option to test the failover process or change the primary write region. Once you add a third region, the **Failover Priorities** option is enabled on the same blade so that you can change the failover order for reads.

Selecting global database regions

There are two common scenarios for configuring two or more regions:

1. Delivering low-latency access to data to end users no matter where they are located around the globe
2. Adding regional resiliency for business continuity and disaster recovery (BCDR)

For delivering low-latency to end-users, it is recommended to deploy both the application and add Azure Cosmos DB in the regions thats correspond to where the application's users are located.

For BCDR, it is recommended to add regions based on the region pairs described in the Business continuity and disaster recovery (BCDR): Azure Paired Regions article.

Connecting to a preferred region using the DocumentDB API

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 Cosmos DB account configuration, current regional availability and the preference list specified, the most optimal endpoint will be chosen by the DocumentDB SDK to perform write and read operations.

This preference list is specified when initializing a connection using the DocumentDB 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 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;

ConnectionPolicy connectionPolicy = new ConnectionPolicy();

//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 Azure Cosmos DB 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.

That's it, that completes this tutorial. You can learn how to manage the consistency of your globally replicated account by reading Consistency levels in Azure Cosmos DB. And for more information about how global database replication works in Azure Cosmos DB, see Distribute data globally with Azure Cosmos DB.

Next steps

In this tutorial, you've done the following:

- Configure global distribution using the Azure portal
- Configure global distribution using the DocumentDB APIs

You can now proceed to the next tutorial to learn how to develop locally using the Azure Cosmos DB local emulator.

Develop locally with the emulator

How to setup Azure Cosmos DB global distribution using the MongoDB API

5/30/2017 • 4 min to read • Edit Online

In this article, we show how to use the Azure portal to setup Azure Cosmos DB global distribution and then connect using the MongoDB API.

This article covers the following tasks:

- Configure global distribution using the Azure portal
- Configure global distribution using the MongoDB API

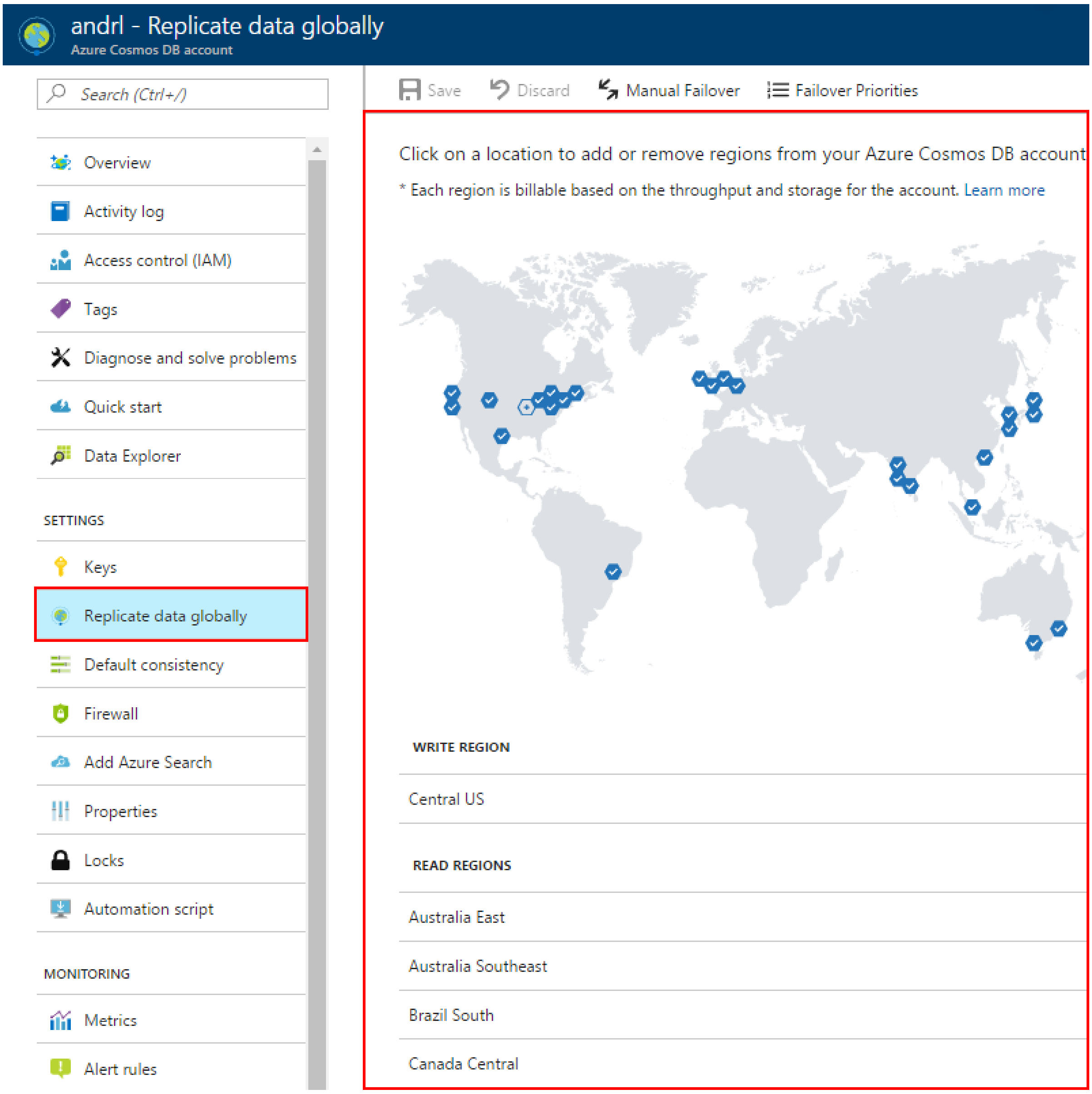
You can learn about Azure Cosmos DB global distribution in this Azure Friday video with Scott Hanselman and Principal Engineering Manager Karthik Raman.

For more information about how global database replication works in Cosmos DB, see Distribute data globally with Cosmos DB.

Add global database regions using the Azure Portal

Azure Cosmos DB is available in all Azure regions world-wide. 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 left bar, click **Azure Cosmos DB**.
2. In the **Azure Cosmos DB** 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 by clicking regions in the map, 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.



Click on a location to add or remove regions from your Azure Cosmos DB account

* Each region is billable based on the throughput and storage for the account. Learn more

WRITE REGION

Central US

READ REGIONS

Australia East

Australia Southeast

Brazil South

Canada Central

Once you add a second region, the **Manual Failover** option is enabled on the **Replicate data globally** blade in the portal. You can use this option to test the failover process or change the primary write region. Once you add a third region, the **Failover Priorities** option is enabled on the same blade so that you can change the failover order for reads.

Selecting global database regions

There are two common scenarios for configuring two or more regions:

1. Delivering low-latency access to data to end users no matter where they are located around the globe
2. Adding regional resiliency for business continuity and disaster recovery (BCDR)

For delivering low-latency to end-users, it is recommended to deploy both the application and add Azure Cosmos DB in the regions thats correspond to where the application's users are located.

For BCDR, it is recommended to add regions based on the region pairs described in the Business continuity and disaster recovery (BCDR): Azure Paired Regions article.

Verifying your regional setup using the MongoDB API

The simplest way of double checking your global configuration within API for MongoDB is to run the *isMaster()* command from the Mongo Shell.

From your Mongo Shell:

```
db.isMaster()
```

Example results:

```
{
  "_t": "IsMasterResponse",
  "ok": 1,
  "ismaster": true,
  "maxMessageSizeBytes": 4194304,
  "maxWriteBatchSize": 1000,
  "minWireVersion": 0,
  "maxWireVersion": 2,
  "tags": {
    "region": "South India"
  },
  "hosts": [
    "vishi-api-for-mongodb-southcentralus.documents.azure.com:10250",
    "vishi-api-for-mongodb-westeurope.documents.azure.com:10250",
    "vishi-api-for-mongodb-southindia.documents.azure.com:10250"
  ],
  "setName": "globaldb",
  "setVersion": 1,
  "primary": "vishi-api-for-mongodb-southindia.documents.azure.com:10250",
  "me": "vishi-api-for-mongodb-southindia.documents.azure.com:10250"
}
```

Connecting to a preferred region using the MongoDB API

The MongoDB API enables you to specify your collection's read preference for a globally distributed database. For both low latency reads and global high availability, we recommend setting your collection's read preference to *nearest*. A read preference of *nearest* is configured to read from the closest region.

```
var collection = database.GetCollection<BsonDocument>(collectionName);
collection = collection.WithReadPreference(new ReadPreference(ReadPreferenceMode.Nearest));
```

For applications with a primary read/write region and a secondary region for disaster recovery (DR) scenarios, we recommend setting your collection's read preference to *secondary preferred*. A read preference of *secondary preferred* is configured to read from the secondary region when the primary region is unavailable.

```
var collection = database.GetCollection<BsonDocument>(collectionName);
collection = collection.WithReadPreference(new ReadPreference(ReadPreferenceMode.SecondaryPreferred));
```

Lastly, if you would like to manually specify your read regions. You can set the region Tag within your read preference.

```
var collection = database.GetCollection<BsonDocument>(collectionName);
var tag = new Tag("region", "Southeast Asia");
collection = collection.WithReadPreference(new ReadPreference(ReadPreferenceMode.Secondary, new[] { new TagSet(new[] { tag }) }));
```

That's it, that completes this tutorial. You can learn how to manage the consistency of your globally replicated account by reading Consistency levels in Azure Cosmos DB. And for more information about how global database replication works in Azure Cosmos DB, see Distribute data globally with Azure Cosmos DB.

Next steps

In this tutorial, you've done the following:

- Configure global distribution using the Azure portal
- Configure global distribution using the DocumentDB APIs

You can now proceed to the next tutorial to learn how to develop locally using the Azure Cosmos DB local emulator.

Develop locally with the emulator

How to setup Azure Cosmos DB global distribution using the Table API

5/30/2017 • 4 min to read • Edit Online

In this article, we show how to use the Azure portal to setup Azure Cosmos DB global distribution and then connect using the Table API (preview).

This article covers the following tasks:

- Configure global distribution using the Azure portal
- Configure global distribution using the Table API

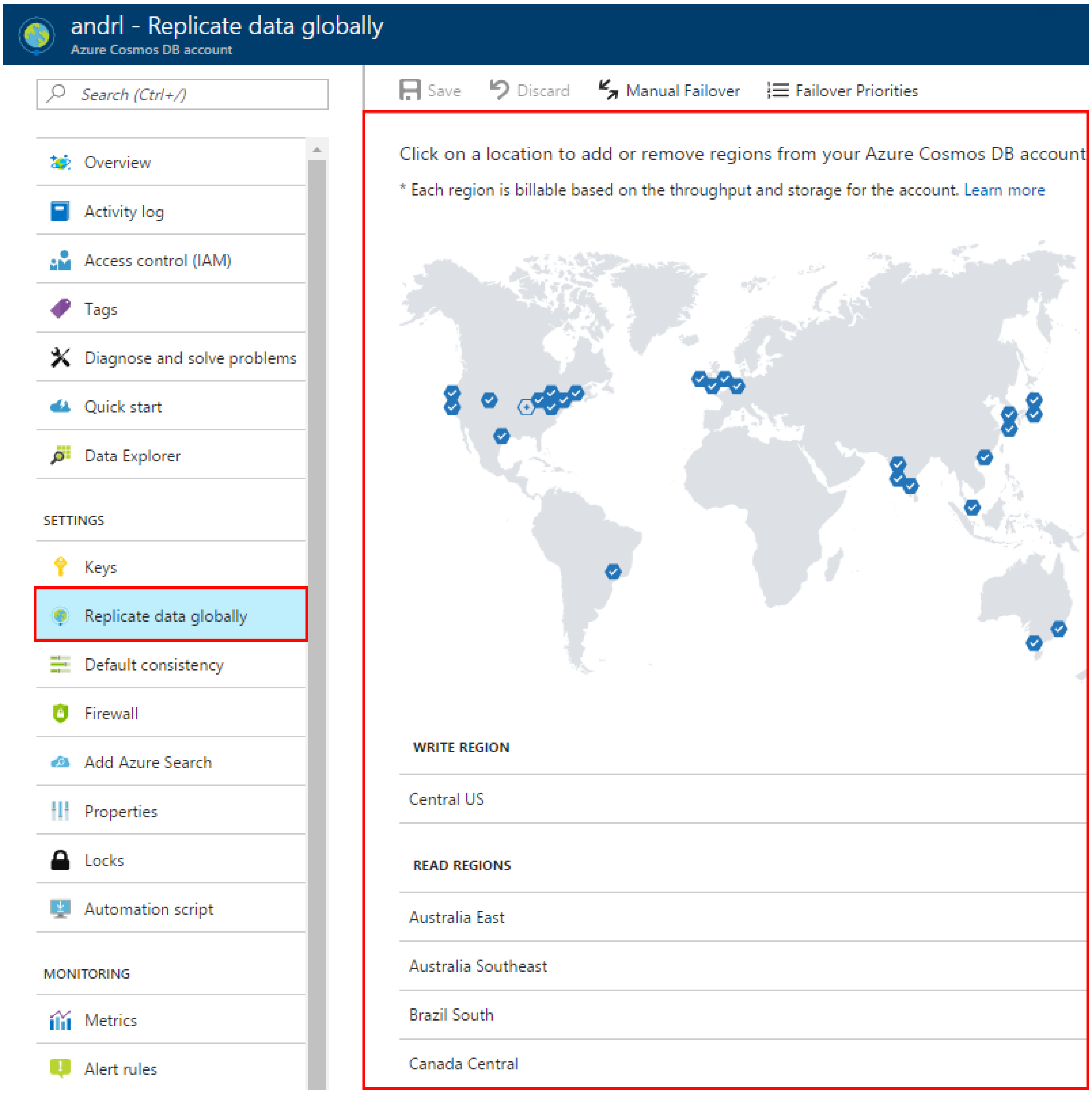
You can learn about Azure Cosmos DB global distribution in this Azure Friday video with Scott Hanselman and Principal Engineering Manager Karthik Raman.

For more information about how global database replication works in Cosmos DB, see Distribute data globally with Cosmos DB.

Add global database regions using the Azure Portal

Azure Cosmos DB is available in all Azure regions world-wide. 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 left bar, click **Azure Cosmos DB**.
2. In the **Azure Cosmos DB** 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 by clicking regions in the map, 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.



Click on a location to add or remove regions from your Azure Cosmos DB account

* Each region is billable based on the throughput and storage for the account. Learn more

WRITE REGION

Central US

READ REGIONS

Australia East

Australia Southeast

Brazil South

Canada Central

Once you add a second region, the **Manual Failover** option is enabled on the **Replicate data globally** blade in the portal. You can use this option to test the failover process or change the primary write region. Once you add a third region, the **Failover Priorities** option is enabled on the same blade so that you can change the failover order for reads.

Selecting global database regions

There are two common scenarios for configuring two or more regions:

1. Delivering low-latency access to data to end users no matter where they are located around the globe
2. Adding regional resiliency for business continuity and disaster recovery (BCDR)

For delivering low-latency to end-users, it is recommended to deploy both the application and add Azure Cosmos DB in the regions thats correspond to where the application's users are located.

For BCDR, it is recommended to add regions based on the region pairs described in the Business continuity and disaster recovery (BCDR): Azure Paired Regions article.

Connecting to a preferred region using the Table API

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 `TablePreferredLocations` configuration value in the app config for the preview Azure Storage SDK. Based on the Azure Cosmos DB account configuration, current regional availability and the preference list specified, the most optimal endpoint will be chosen by the Azure Storage SDK to perform write and read operations.

The `TablePreferredLocations` should contain a comma-separated list of preferred (multi-homing) locations for reads. Each client instance can specify a subset of these regions in the preferred order for low latency reads. The regions must be named using their display names, for example, `West US`.

All reads will be sent to the first available region in the `TablePreferredLocations` list. If the request fails, the client will fail down the list to the next region, and so on.

The SDK will only attempt to read from the regions specified in `TablePreferredLocations`. So, for example, if the Database Account is available in three regions, but the client only specifies two of the non-write regions for `TablePreferredLocations`, then no reads will be served out of the write region, even in the case of failover.

The SDK will automatically send all writes to the current write region.

If the `TablePreferredLocations` property is not set, all requests will be served from the current write region.

```
<appSettings>
  <add key="TablePreferredLocations" value="East US, West US, North Europe"/>
</appSettings>
```

That's it, that completes this tutorial. You can learn how to manage the consistency of your globally replicated account by reading Consistency levels in Azure Cosmos DB. And for more information about how global database replication works in Azure Cosmos DB, see Distribute data globally with Azure Cosmos DB.

Next steps

In this tutorial, you've done the following:

- Configure global distribution using the Azure portal
- Configure global distribution using the DocumentDB APIs

You can now proceed to the next tutorial to learn how to develop locally using the Azure Cosmos DB local emulator.

Develop locally with the emulator

How to setup Azure Cosmos DB global distribution using the Graph API

5/30/2017 • 5 min to read • Edit Online

In this article, we show how to use the Azure portal to setup Azure Cosmos DB global distribution and then connect using the Graph API (preview).

This article covers the following tasks:

- Configure global distribution using the Azure portal
- Configure global distribution using the Graph APIs (preview)

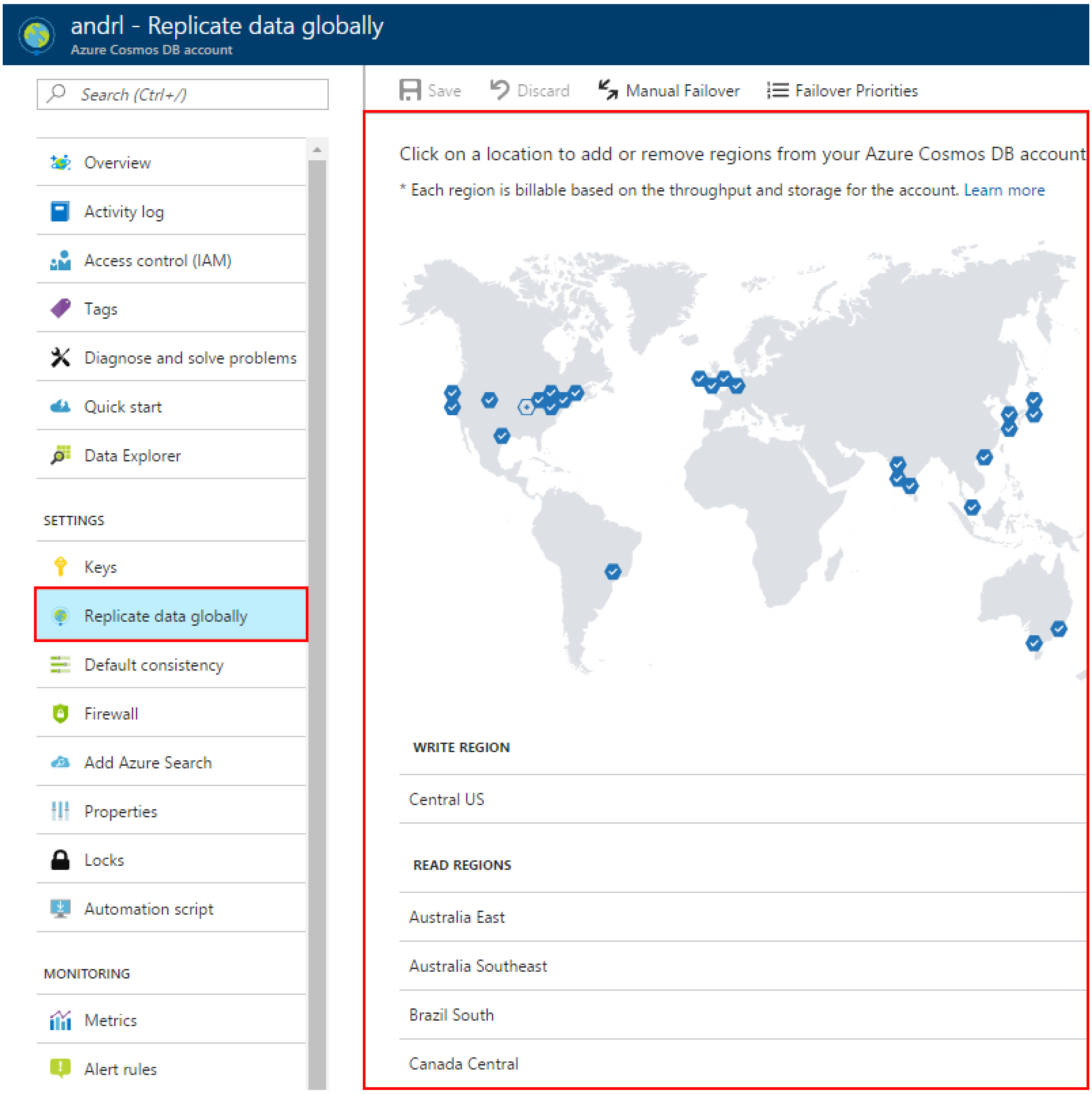
You can learn about Azure Cosmos DB global distribution in this Azure Friday video with Scott Hanselman and Principal Engineering Manager Karthik Raman.

For more information about how global database replication works in Cosmos DB, see Distribute data globally with Cosmos DB.

Add global database regions using the Azure Portal

Azure Cosmos DB is available in all Azure regions world-wide. 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 left bar, click **Azure Cosmos DB**.
2. In the **Azure Cosmos DB** 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 by clicking regions in the map, 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.



Click on a location to add or remove regions from your Azure Cosmos DB account

* Each region is billable based on the throughput and storage for the account. Learn more

WRITE REGION

Central US

READ REGIONS

Australia East

Australia Southeast

Brazil South

Canada Central

Once you add a second region, the **Manual Failover** option is enabled on the **Replicate data globally** blade in the portal. You can use this option to test the failover process or change the primary write region. Once you add a third region, the **Failover Priorities** option is enabled on the same blade so that you can change the failover order for reads.

Selecting global database regions

There are two common scenarios for configuring two or more regions:

1. Delivering low-latency access to data to end users no matter where they are located around the globe
2. Adding regional resiliency for business continuity and disaster recovery (BCDR)

For delivering low-latency to end-users, it is recommended to deploy both the application and add Azure Cosmos DB in the regions thats correspond to where the application's users are located.

For BCDR, it is recommended to add regions based on the region pairs described in the Business continuity and disaster recovery (BCDR): Azure Paired Regions article.

Connecting to a preferred region using the Graph API using the .NET SDK

The Graph API is exposed as an extension library on top of the DocumentDB SDK.

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 Cosmos DB 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 SDKs. The SDKs accept an optional parameter "PreferredLocations" that is an ordered list of Azure regions.

- **Writes**: The SDK will automatically send all writes to the current write region.
- **Reads**: 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 SDKs will only attempt to read from the regions specified in PreferredLocations. So, for example, if the Cosmos DB 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.

Using the SDK

For example, in the .NET SDK, the `ConnectionPolicy` parameter for the `DocumentClient` constructor has a property called `PreferredLocations`. This property can be set to a list of region names. The display names for Azure Regions can be specified as part of `PreferredLocations`.

> 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;

ConnectionPolicy connectionPolicy = new ConnectionPolicy();

//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 Azure Cosmos DB
await docClient.OpenAsync().ConfigureAwait(false);
```

That's it, that completes this tutorial. You can learn how to manage the consistency of your globally replicated account by reading Consistency levels in Azure Cosmos DB. And for more information about how global database replication works in Azure Cosmos DB, see Distribute data globally with Azure Cosmos DB.

Next steps

In this tutorial, you've done the following:

- Configure global distribution using the Azure portal

- Configure global distribution using the DocumentDB APIs

You can now proceed to the next tutorial to learn how to develop locally using the Azure Cosmos DB local emulator.

Develop locally with the emulator

Use the Azure Cosmos DB Emulator for local development and testing

6/9/2017 • 11 min to read • Edit Online

Binaries	Download MSI
Docker	Docker Hub
Docker source	Github

The Azure Cosmos DB Emulator provides a local environment that emulates the Azure Cosmos DB service for development purposes. Using the Azure Cosmos DB 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 Azure Cosmos DB Emulator, you can switch to using an Azure Cosmos DB account in the cloud.

This article covers the following tasks:

- Installing the Emulator
- Running the Emulator on Docker for Windows
- Authenticating requests
- Using the Data Explorer in the Emulator
- Exporting SSL certificates
- Calling the Emulator from the command line
- Collecting trace files

We recommend getting started by watching the following video, where Kirill Gavrylyuk shows how to get started with the Azure Cosmos DB Emulator. Note that the video refers to the emulator as the DocumentDB Emulator, but the tool itself has been renamed the Azure Cosmos DB Emulator since taping the video. All information in the video is still accurate for the Azure Cosmos DB Emulator.

How the Emulator works

The Azure Cosmos DB Emulator provides a high-fidelity emulation of the Azure Cosmos DB service. It supports identical functionality as Azure Cosmos DB, 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 Azure Cosmos DB Emulator, and deploy them to Azure at global scale by just making a single configuration change to the connection endpoint for Azure Cosmos DB.

While we created a high-fidelity local emulation of the actual Azure Cosmos DB service, the implementation of the Azure Cosmos DB Emulator is different than that of the service. For example, the Azure Cosmos DB 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 Azure Cosmos DB Emulator.

> NOTE
>
> At this time the Data Explorer in the emulator only supports the creation of DocumentDB API collections and MongoDB collections. The Data Explorer in the emulator does not currently support the creation of tables and graphs.

System requirements

The Azure Cosmos DB 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

Installation

You can download and install the Azure Cosmos DB Emulator from the Microsoft Download Center.

> NOTE
>
> To install, configure, and run the Azure Cosmos DB Emulator, you must have administrative privileges on the computer.

Running on Docker for Windows

The Azure Cosmos DB Emulator can be run on Docker for Windows. The Emulator does not work on Docker for Oracle Linux.

Once you have Docker for Windows installed, you can pull the Emulator image from Docker Hub by running the following command from your favorite shell (cmd.exe, PowerShell, etc.).

```
docker pull microsoft/azure-cosmosdb-emulator
```

To start the image, run the following commands.

```
md %LOCALAPPDATA%\CosmosDBEmulatorCert 2>nul
docker run -v %LOCALAPPDATA%\CosmosDBEmulatorCert:c:\CosmosDBEmulator\CosmosDBEmulatorCert -P -t -i microsoft/azure-cosmosdb-emulator
```

The response looks similar to the following:

```
Starting Emulator
Emulator Endpoint: https://172.20.229.193:8081/
Master Key: C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==
Exporting SSL Certificate
You can import the SSL certificate from an administrator command prompt on the host by running:
cd /d %LOCALAPPDATA%\CosmosDBEmulatorCert
powershell .\importcert.ps1
--------------------------------------------------------------------------------------------------
Starting interactive shell
```

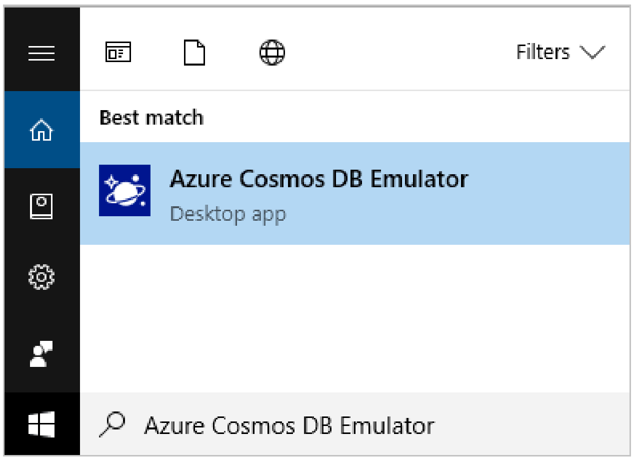
Closing the interactive shell once the Emulator has been started will shutdown the Emulator's container.

Use the endpoint and master key in from the response in your client and import the SSL certificate into your host. To import the SSL certificate, do the following from an admin command prompt:

```
cd %LOCALAPPDATA%\CosmosDBEmulatorCert
powershell .\importcert.ps1
```

Start the Emulator

To start the Azure Cosmos DB Emulator, select the Start button or press the Windows key. Begin typing **Azure Cosmos DB Emulator**, and select the emulator from the list of applications.



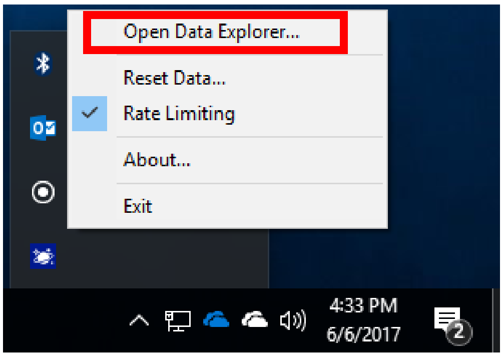
When the emulator is running, you'll see an icon in the Windows taskbar notification area.

The Azure Cosmos DB Emulator by default runs on the local machine ("localhost") listening on port 8081.

The Azure Cosmos DB Emulator is installed by default to the `C:\Program Files\Azure Cosmos DB Emulator` directory. You can also start and stop the emulator from the command-line. See command-line tool reference for more information.

Start Data Explorer

When the Azure Cosmos DB emulator launches it will automatically open the Azure Cosmos DB Data Explorer in your browser. The address will appear as https://localhost:8081/_explorer/index.html. If you close the explorer and would like to re-open it later, you can either open the URL in your browser or launch it from the Azure Cosmos DB Emulator in the Windows Tray Icon as shown below.



Checking for updates

Data Explorer indicates if there is a new update available for download.

> NOTE
>
> Data created in one version of the Azure Cosmos DB 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 Cosmos DB account, rather than in the Azure Cosmos DB Emulator.

Authenticating requests

Just as with Azure Document in the cloud, every request that you make against the Azure Cosmos DB Emulator must be authenticated. The Azure Cosmos DB 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 Azure Cosmos DB Emulator. They are:

```
Account name: localhost:<port>
Account key: C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==
```

> NOTE
>
> The master key supported by the Azure Cosmos DB Emulator is intended for use only with the emulator. You cannot use your production Azure Cosmos DB account and key with the Azure Cosmos DB Emulator.

Additionally, just as the Azure Cosmos DB service, the Azure Cosmos DB Emulator supports only secure communication via SSL.

Developing with the Emulator

Once you have the Azure Cosmos DB Emulator running on your desktop, you can use any supported Azure Cosmos DB SDK or the Azure Cosmos DB REST API to interact with the Emulator. The Azure Cosmos DB Emulator also includes a built-in Data Explorer that lets you create collections for the DocumentDB and MongoDB APIs, and view and edit documents without writing any code.

```
// Connect to the Azure Cosmos DB Emulator running locally
DocumentClient client = new DocumentClient(
   new Uri("https://localhost:8081"),
   "C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==");
```

If you're using Azure Cosmos DB protocol support for MongoDB, please use the following connection string:

```
mongodb://localhost:C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw
==@localhost:10250/admin?ssl=true&3t.sslSelfSignedCerts=true
```

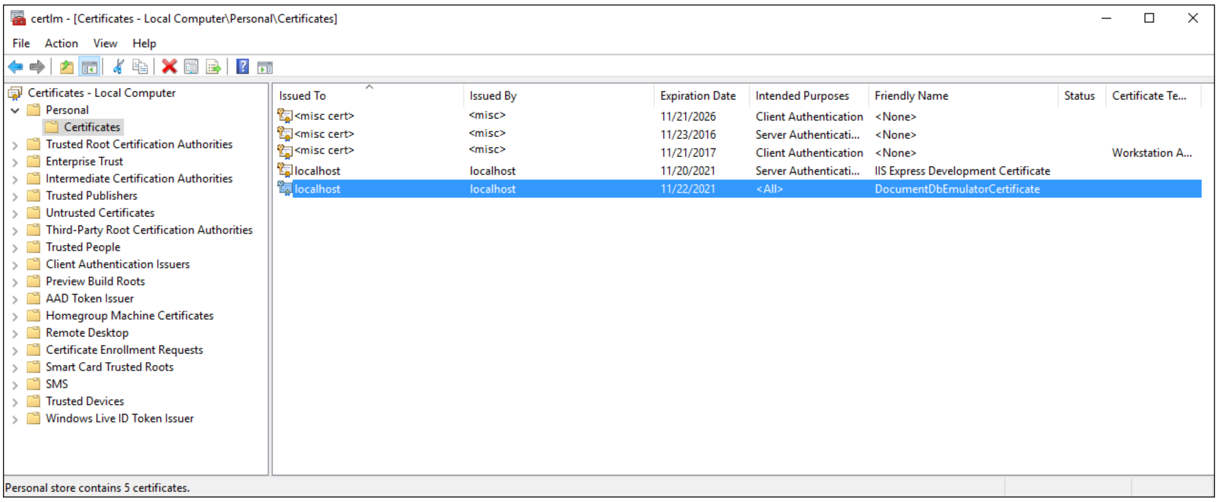
You can use existing tools like Azure DocumentDB Studio to connect to the Azure Cosmos DB Emulator. You can also migrate data between the Azure Cosmos DB Emulator and the Azure Cosmos DB service using the Azure Cosmos DB Data Migration Tool.

Using the Azure Cosmos DB emulator, by default, you can create up to 25 single partition collections or 1 partitioned collection. For more information about changing this value, see Setting the PartitionCount value.

Export the SSL certificate

.NET languages and runtime use the Windows Certificate Store to securely connect to the Azure Cosmos DB local emulator. Other languages have their own method of managing and using certificates. Java uses its own certificate store whereas Python uses socket wrappers.

In order to obtain a certificate to use with languages and runtimes that do not integrate with the Windows Certificate Store you will need to export it using the Windows Certificate Manager. You can start it by running certlm.msc or follow the step by step instructions in Export the Azure Cosmos DB Emulator Certificates. Once the certificate manager is running, open the Personal Certificates as shown below and export the certificate with the friendly name "DocumentDBEmulatorCertificate" as a BASE-64 encoded X.509 (.cer) file.


certlm - [Certificates - Local Computer\Personal\Certificates]
File
Action
View
Help

- Certificates - Local Computer
  - Personal
    - Certificates
  - Trusted Root Certification Authorities
  - Enterprise Trust
  - Intermediate Certification Authorities
  - Trusted Publishers
  - Untrusted Certificates
  - Third-Party Root Certification Authorities
  - Trusted People
  - Client Authentication Issuers
  - Preview Build Roots
  - AAD Token Issuer
  - Homegroup Machine Certificates
  - Remote Desktop
  - Certificate Enrollment Requests
  - Smart Card Trusted Roots
  - SMS
  - Trusted Devices
  - Windows Live ID Token Issuer

Issued To	Issued By	Expiration Date	Intended Purposes	Friendly Name	Status	Certificate Te...
<misc cert>	<misc>	11/21/2026	Client Authentication	<None>		
<misc cert>	<misc>	11/23/2016	Server Authenticati...	<None>		
<misc cert>	<misc>	11/21/2017	Client Authentication	<None>		Workstation A...
localhost	localhost	11/20/2021	Server Authenticati...	IIS Express Development Certificate		
localhost	localhost	11/22/2021	<All>	DocumentDbEmulatorCertificate		

Personal store contains 5 certificates.

The X.509 certificate can be imported into the Java certificate store by following the instructions in Adding a Certificate to the Java CA Certificates Store. Once the certificate is imported into the certificate store, Java and MongoDB applications will be able to connect to the Azure Cosmos DB Emulator.

When connecting to the emulator from Python and Node.js SDKs, SSL verification is disabled.

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

```
CosmosDB.Emulator.exe [/Shutdown] [/DataPath] [/Port] [/MongoPort] [/DirectPorts] [/Key] [/EnableRateLimiting] [/DisableRateLimiting]
[/NoUI] [/NoExplorer] [/?]
```

To view the list of options, type `CosmosDB.Emulator.exe /?` at the command prompt.

Option	Description	Command	Arguments
[No arguments]	Starts up the Azure Cosmos DB Emulator with default settings.	CosmosDB.Emulator.exe	
[Help]	Displays the list of supported command-line arguments.	CosmosDB.Emulator.exe /?	
Shutdown	Shuts down the Azure Cosmos DB Emulator.	CosmosDB.Emulator.exe /Shutdown	
DataPath	Specifies the path in which to store data files. Default is %LocalAppdata%\CosmosDBEmulator.	CosmosDB.Emulator.exe /DataPath=<datapath>	<datapath>: An accessible path
Port	Specifies the port number to use for the emulator. Default is 8081.	CosmosDB.Emulator.exe /Port=<port>	<port>: Single port number
MongoPort	Specifies the port number to use for MongoDB compatibility API. Default is 10250.	CosmosDB.Emulator.exe /MongoPort=<mongoport>	<mongoport>: Single port number
DirectPorts	Specifies the ports to use for direct connectivity. Defaults are 10251,10252,10253,10254.	CosmosDB.Emulator.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.	CosmosDB.Emulator.exe /Key:<key>	<key>: Key must be the base-64 encoding of a 64-byte vector
EnableRateLimiting	Specifies that request rate limiting behavior is enabled.	CosmosDB.Emulator.exe /EnableRateLimiting	
DisableRateLimiting	Specifies that request rate limiting behavior is disabled.	CosmosDB.Emulator.exe /DisableRateLimiting	
NoUI	Do not show the emulator user interface.	CosmosDB.Emulator.exe /NoUI	
NoExplorer	Don't show document explorer on startup.	CosmosDB.Emulator.exe /NoExplorer	
PartitionCount	Specifies the maximum number of partitioned collections. See Change the number of collections for more information.	CosmosDB.Emulator.exe /PartitionCount=<partitioncount>	<partitioncount>: Maximum number of allowed single partition collections. Default is 25. Maximum allowed is 250.

Differences between the Azure Cosmos DB Emulator and Azure Cosmos DB

Because the Azure Cosmos DB Emulator provides an emulated environment running on a local developer workstation, there are some differences in functionality between the emulator and an Azure Cosmos DB account in the cloud:

- The Azure Cosmos DB Emulator supports only a single fixed account and a well-known master key. Key regeneration is not possible in the Azure Cosmos DB Emulator.
- The Azure Cosmos DB Emulator is not a scalable service and will not support a large number of collections.
- The Azure Cosmos DB Emulator does not simulate different Azure Cosmos DB consistency levels.
- The Azure Cosmos DB Emulator does not simulate multi-region replication.
- The Azure Cosmos DB Emulator does not support the service quota overrides that are available in the Azure Cosmos DB service (e.g. document size limits, increased partitioned collection storage).
- As your copy of the Azure Cosmos DB Emulator might not be up to date with the most recent changes with the Azure Cosmos DB service, please Azure Cosmos DB capacity planner to accurately estimate production throughput (RUs) needs of your application.

Change the number of collections

By default, you can create up to 25 single partition collections, or 1 partitioned collection using the Azure Cosmos DB Emulator. By modifying the **PartitionCount** value, you can create up to 250 single partition collections or 10 partitioned collections, or any combination of the two that does not exceed 250 single partitions (where 1 partitioned collection = 25 single partition collection).

If you attempt to create a collection after the current partition count has been exceeded, the emulator throws a ServiceUnavailable exception, with the following message.

```
Sorry, we are currently experiencing high demand in this region,
and cannot fulfill your request at this time. We work continuously
to bring more and more capacity online, and encourage you to try again.
Please do not hesitate to email docdbswat@microsoft.com at any time or
for any reason. ActivityId: 29da65cc-fba1-45f9-b82c-bf01d78a1f91
```

To change the number of collections available to the Azure Cosmos DB Emulator, do the following:

1. Delete all local Azure Cosmos DB Emulator data by right-clicking the **Azure Cosmos DB Emulator** icon on the system tray, and then clicking **Reset Data…**.
2. Delete all emulator data in this folder C:\Users\user_name\AppData\Local\CosmosDBEmulator.
3. Exit all open instances by right-clicking the **Azure Cosmos DB Emulator** icon on the system tray, and then clicking **Exit**. It may take a minute for all instances to exit.
4. Install the latest version of the Azure Cosmos DB Emulator.
5. Launch the emulator with the PartitionCount flag by setting a value <= 250. For example: `C:\Program Files\Azure Cosmos DB Emulator>CosmosDB.Emulator.exe /PartitionCount=100`.

Troubleshooting

Use the following tips to help troubleshoot issues you encounter with the Azure Cosmos DB emulator:

- If the Azure Cosmos DB emulator crashes, collect dump files from c:\Users\user_name\AppData\Local\CrashDumps folder, compress them, and attach them to an email to askcosmosdb@microsoft.com.

- If you experience crashes in CosmosDB.StartupEntryPoint.exe, run the following command from an admin command prompt: `lodctr /R`
- If you encounter a connectivity issue, collect trace files, compress them, and attach them to an email to askcosmosdb@microsoft.com.
- If you receive a **Service Unavailable** message, the emulator might be failing to initialize the network stack. Check to see if you have the Pulse secure client or Juniper networks client installed, as their network filter drivers may cause the problem. Uninstalling third party network filter drivers typically fixes the issue.

Collect trace files

To collect debugging traces, run the following commands from an administrative command prompt:

1. `cd /d "%ProgramFiles%\Azure Cosmos DB Emulator"`
2. `CosmosDB.Emulator.exe /shutdown`. Watch the system tray to make sure the program has shut down, it may take a minute. You can also just click **Exit** in the Azure Cosmos DB emulator user interface.
3. `CosmosDB.Emulator.exe /starttraces`
4. `CosmosDB.Emulator.exe`
5. Reproduce the problem. If Data Explorer is not working, you only need to wait for the browser to open for a few seconds to catch the error.
6. `CosmosDB.Emulator.exe /stoptraces`
7. Navigate to `%ProgramFiles%\Azure Cosmos DB Emulator` and find the docdbemulator_000001.etl file.
8. Send the .etl file along with repro steps to askcosmosdb@microsoft.com for debugging.

Next steps

In this tutorial, you've done the following:

- Installed the local Emulator
- Rand the Emulator on Docker for Windows
- Authenticated requests
- Used the Data Explorer in the Emulator
- Exported SSL certificates
- Called the Emulator from the command line
- Collected trace files

In this tutorial, you've learned how to use the local Emulator for free local development. You can now proceed to the next tutorial and learn how to export Emulator SSL certificates.

Export the Azure Cosmos DB Emulator certificates

Export the Azure Cosmos DB Emulator certificates for use with Java, Python, and Node.js

6/9/2017 • 3 min to read • Edit Online

Download the Emulator

The Azure Cosmos DB Emulator provides a local environment that emulates the Azure Cosmos DB service for development purposes including its use of SSL connections. This post demonstrates how to export the SSL certificates for use in languages and runtimes that do not integrate with the Windows Certificate Store such as Java which uses its own certificate store and Python which uses socket wrappers and Node.js which uses tlsSocket. You can read more about the emulator in Use the Azure Cosmos DB Emulator for development and testing.

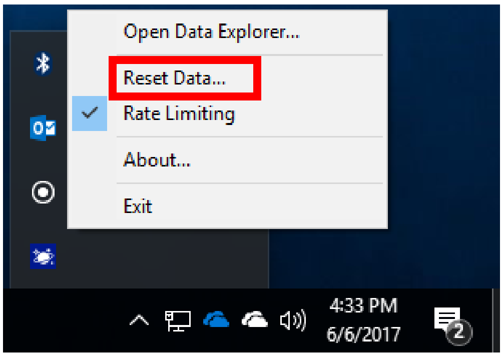
This tutorial covers the following tasks:

- Rotating certificates
- Exporting SSL certificate
- Learning how to use the certificate in Java, Python, and Node.js

Certification rotation

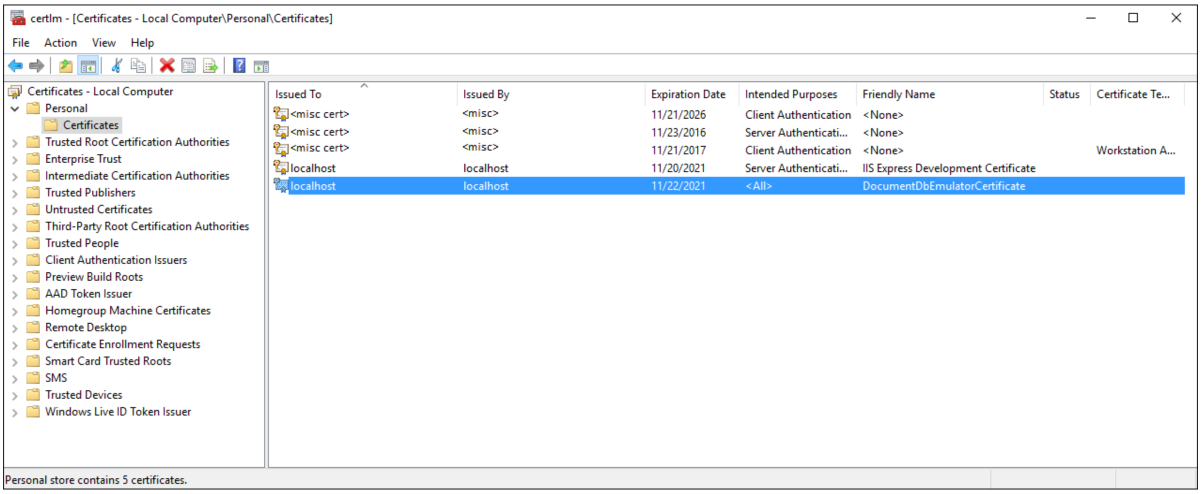
Certificates in the Azure Cosmos DB Local Emulator are generated the first time the emulator is run. There are two certificates. One used for connecting to the local emulator and one for managing secrets within the emulator. The certificate you want to export is the connection certificate with the friendly name "DocumentDBEmulatorCertificate".

Both certificates can be regenerated by clicking **Reset Data** as shown below from Azure Cosmos DB Emulator running in the Windows Tray. If you regenerate the certificates and have installed them into the Java certificate store or used them elsewhere you will need to update them, otherwise your application will no longer connect to the local emulator.



How to export the Azure Cosmos DB SSL certificate

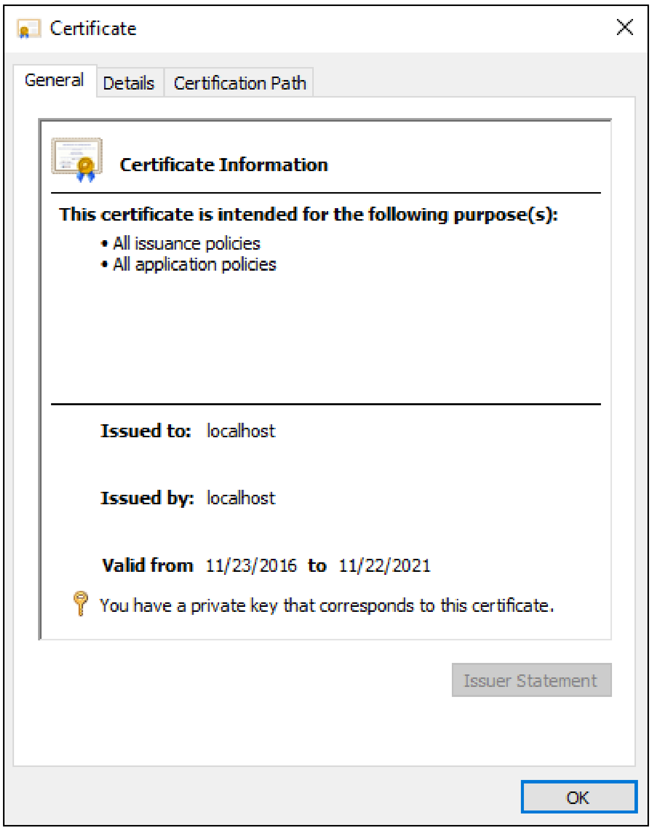
1. Start the Windows Certificate manager by running certlm.msc and navigate to the Personal->Certificates folder and open the certificate with the friendly name **DocumentDbEmulatorCertificate**.


certlm - [Certificates - Local Computer\Personal\Certificates]
File
Action
View
Help
Certificates - Local Computer
Personal
Certificates
Trusted Root Certification Authorities
Enterprise Trust
Intermediate Certification Authorities
Trusted Publishers
Untrusted Certificates
Third-Party Root Certification Authorities
Trusted People
Client Authentication Issuers
Preview Build Roots
AAD Token Issuer
Homegroup Machine Certificates
Remote Desktop
Certificate Enrollment Requests
Smart Card Trusted Roots
SMS
Trusted Devices
Windows Live ID Token Issuer

Issued To	Issued By	Expiration Date	Intended Purposes	Friendly Name	Status	Certificate Te...
<misc cert>	<misc>	11/21/2026	Client Authentication	<None>		
<misc cert>	<misc>	11/23/2016	Server Authenticati...	<None>		
<misc cert>	<misc>	11/21/2017	Client Authentication	<None>		Workstation A...
localhost	localhost	11/20/2021	Server Authenticati...	IIS Express Development Certificate		
localhost	localhost	11/22/2021	<All>	DocumentDbEmulatorCertificate		

Personal store contains 5 certificates.

2. Click on **Details** then **OK**.



Certificate Information

This certificate is intended for the following purpose(s):

- All issuance policies
- All application policies

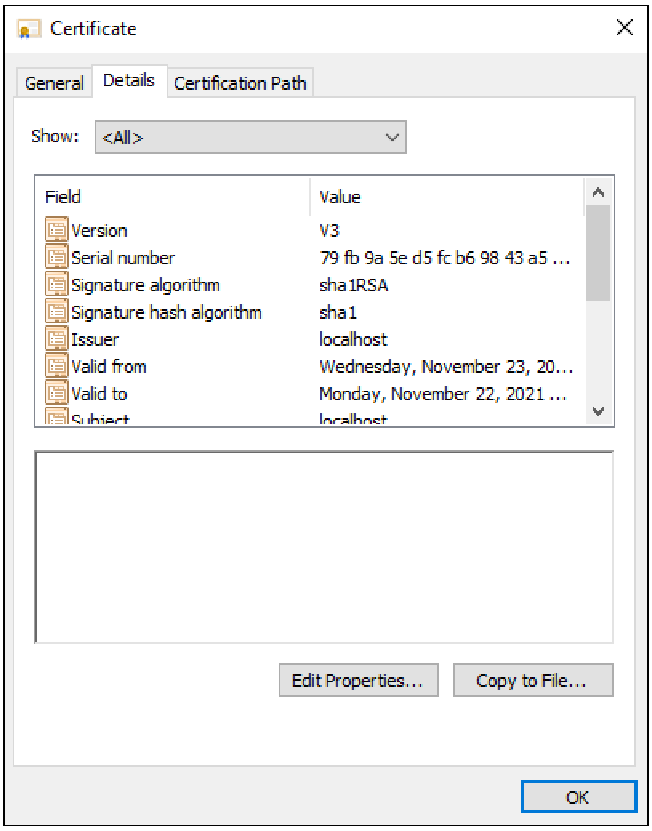
Issued to: localhost

Issued by: localhost

Valid from 11/23/2016 **to** 11/22/2021

You have a private key that corresponds to this certificate.

3. Click **Copy to File...**.

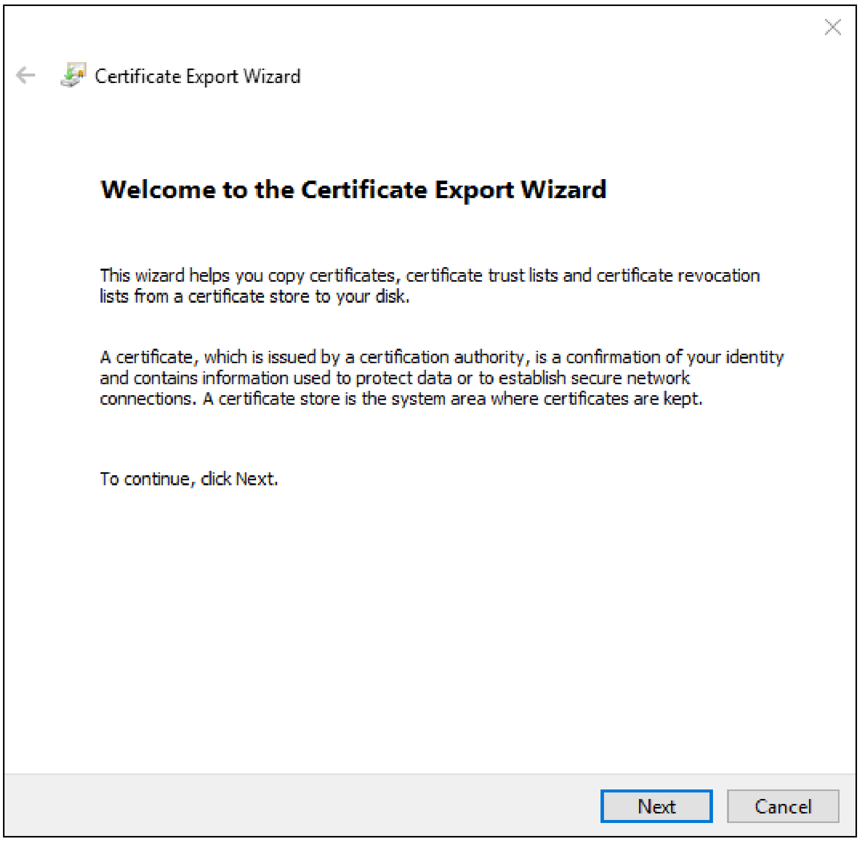


Certificate

Show: <All>

Field	Value
Version	V3
Serial number	79 fb 9a 5e d5 fc b6 98 43 a5 ...
Signature algorithm	sha1RSA
Signature hash algorithm	sha1
Issuer	localhost
Valid from	Wednesday, November 23, 20...
Valid to	Monday, November 22, 2021 ...
Subject	localhost

4. Click **Next**.



Certificate Export Wizard

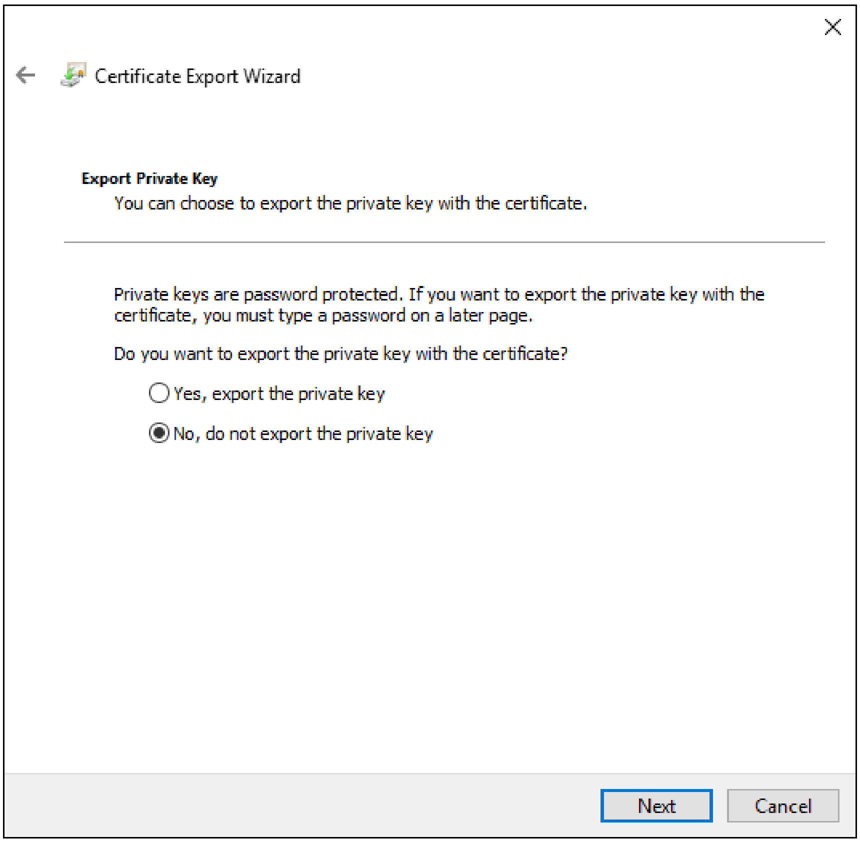
Welcome to the Certificate Export Wizard

This wizard helps you copy certificates, certificate trust lists and certificate revocation lists from a certificate store to your disk.

A certificate, which is issued by a certification authority, is a confirmation of your identity and contains information used to protect data or to establish secure network connections. A certificate store is the system area where certificates are kept.

To continue, click Next.

5. Click **No, do not export private key**, then click **Next**.



Certificate Export Wizard

Export Private Key
You can choose to export the private key with the certificate.

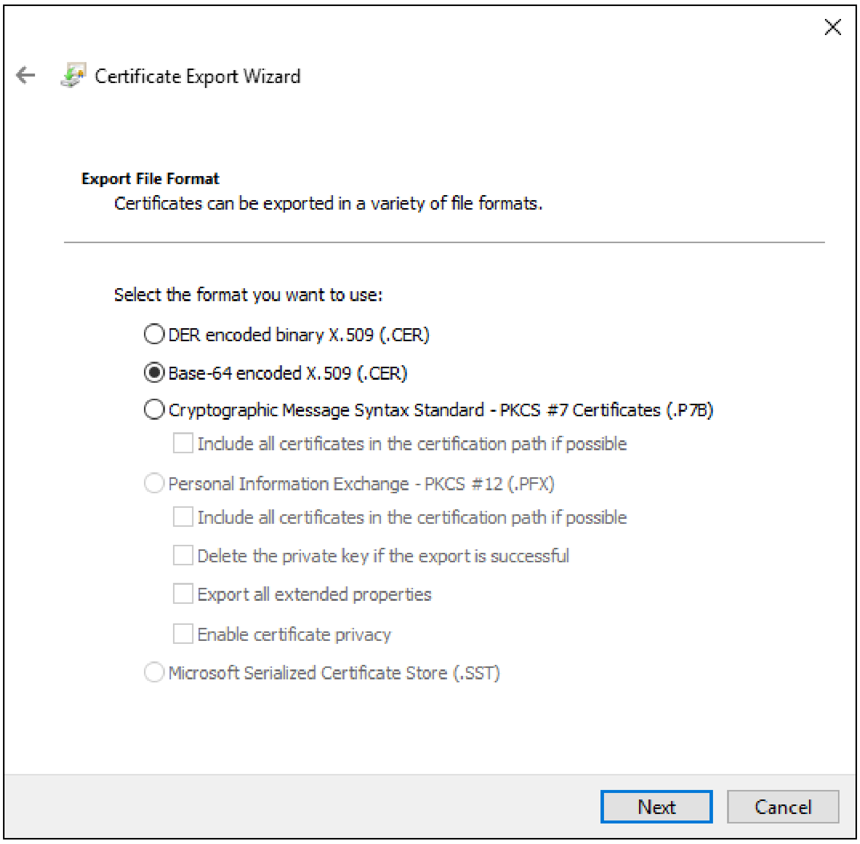
Private keys are password protected. If you want to export the private key with the certificate, you must type a password on a later page.

Do you want to export the private key with the certificate?

○ Yes, export the private key

● No, do not export the private key

6. Click on **Base-64 encoded X.509 (.CER)** and then **Next**.



Certificate Export Wizard

Export File Format
Certificates can be exported in a variety of file formats.

Select the format you want to use:

○ DER encoded binary X.509 (.CER)

● Base-64 encoded X.509 (.CER)

○ Cryptographic Message Syntax Standard - PKCS #7 Certificates (.P7B)

□ Include all certificates in the certification path if possible

○ Personal Information Exchange - PKCS #12 (.PFX)

□ Include all certificates in the certification path if possible

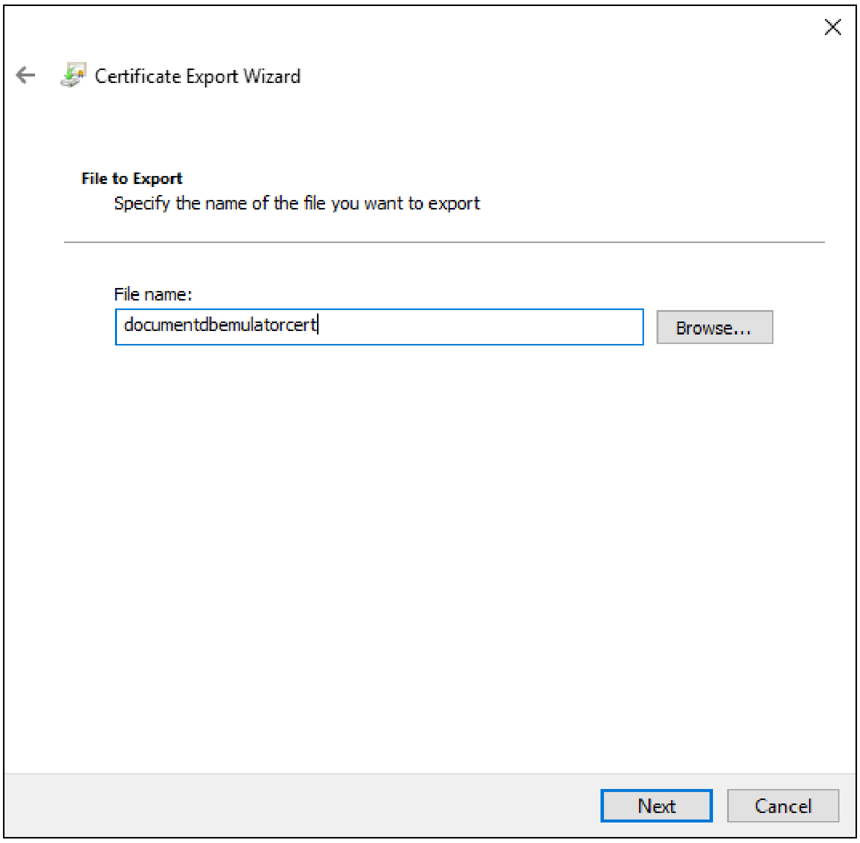
□ Delete the private key if the export is successful

□ Export all extended properties

□ Enable certificate privacy

○ Microsoft Serialized Certificate Store (.SST)

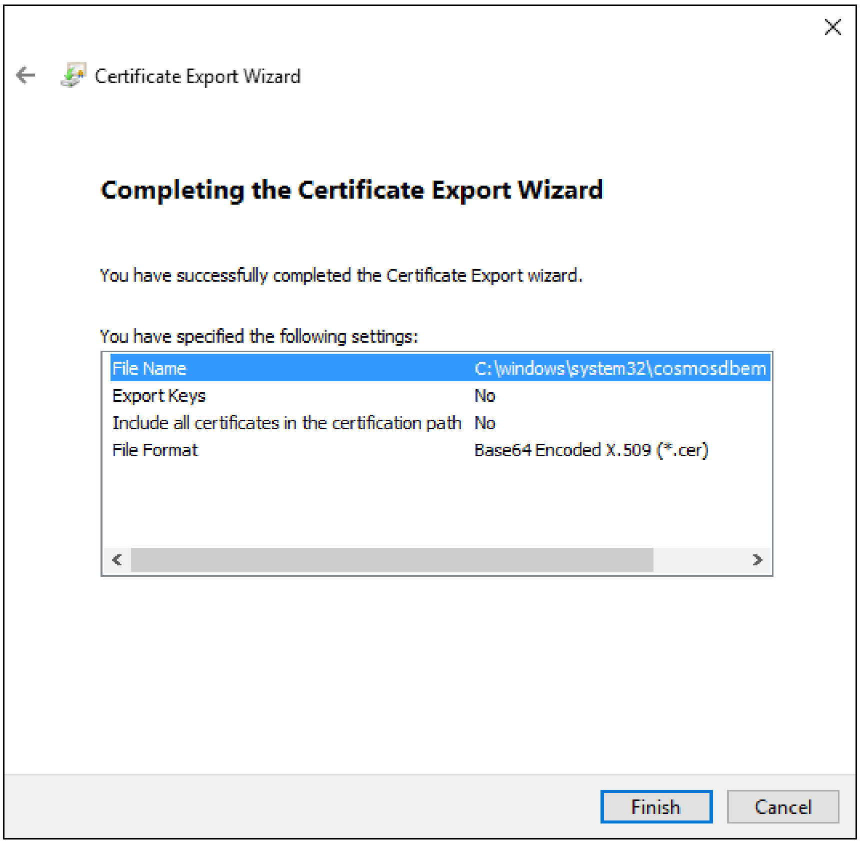
7. Give the certificate a name. In this case **documentdbemulatorcert** and then click **Next**.



File to Export
Specify the name of the file you want to export

File name:
documentdbemulatorcert

8. Click **Finish**.



Completing the Certificate Export Wizard

You have successfully completed the Certificate Export wizard.

You have specified the following settings:

File Name	C:\windows\system32\cosmosdbem
Export Keys	No
Include all certificates in the certification path	No
File Format	Base64 Encoded X.509 (*.cer)

How to use the certificate in Java

When running Java applications or MongoDB applications that use the Java client it is easier to install the certificate into the Java default certificate store than passing the "-Djavax.net.ssl.trustStore= -Djavax.net.ssl.trustStorePassword="" flags. For example the included Java Demo application depends on the default certificate store.

Follow the instructions in the Adding a Certificate to the Java CA Certificates Store to import the X.509 certificate into the default Java certificate store. Keep in mind you will be working in the %JAVA_HOME% directory when running keytool.

Once the "CosmosDBEmulatorCertificate" SSL certificate is installed your application should be able to connect and use the local Azure Cosmos DB Emulator. If you continue to have trouble you may want to follow the Debugging SSL/TLS Connections article. It is very likely the certificate is not installed into the %JAVA_HOME%/jre/lib/security/cacerts store. For example if you have multiple installed versions of Java your application may be using a different cacerts store than the one you updated.

How to use the certificate in Python

By default the Python SDK(version 2.0.0 or higher) for the DocumentDB API will not try and use the SSL certificate when connecting to the local emulator. If however you want to use SSL validation you can follow the examples in the Python socket wrappers documentation.

How to use the certificate in Node.js

By default the Node.js SDK(version 1.10.1 or higher) for the DocumentDB API will not try and use the SSL certificate when connecting to the local emulator. If however you want to use SSL validation you can follow the examples in the Node.js documentation.

Next steps

In this tutorial, you've done the following:

- Rotated certificates
- Exported the SSL certificate
- Learned how to use the certificate in Java, Python and Node.js

You can now proceed to the Concepts section for more information about Cosmos DB.

Global distribution

Azure CLI samples for Azure Cosmos DB

6/7/2017 • 1 min to read • Edit Online

The following table includes links to sample Azure CLI scripts for Azure Cosmos DB. Reference pages for all Azure Cosmos DB CLI commands are available in the Azure CLI 2.0 Reference.

Create Azure Cosmos DB account, database, and containers	
Create a DocumentDB, Graph, or Table API account	Creates a single Azure Cosmos DB API account, database, and container for use with the DocumentDB, Graph, or Table APIs.
Create a MongoDB API account	Creates a single Azure Cosmos DB MongoDB API account, database, and collection.
Scale Azure Cosmos DB	
Scale container throughput	Changes the provisioned througput on a container.
Replicate Azure Cosmos DB database account in multiple regions and configure failover priorities	Globally replicates account data into multiple regions with a specified failover priority.
Secure Azure Cosmos DB	
Get account keys	Gets the primary and secondary master write keys and primary and secondary read-only keys for the account.
Get MongoDB connection string	Gets the connection string to connect your MongoDB app to your Azure Cosmos DB account.
Regenerate account keys	Regenerates the master or read-only key for the account.
Create a firewall	Creates an inbound IP access control policy to limit access to the account from an approved set of machines and/or cloud services.
High availability, disaster recovery, backup and restore	
Configure failover policy	Sets the failover priority of each region in which the account is replicated.
Connect Azure Cosmos DB to resources	
Connect a web app to Azure Cosmos DB	Create and connect an Azure Cosmos DB database and an Azure web app.

Azure PowerShell samples for Azure Cosmos DB

6/1/2017 • 1 min to read • Edit Online

The following table includes links to sample Azure PowerShell scripts for Azure Cosmos DB.

Create an Azure Cosmos DB account	
Create a DocumentDB API account	Creates a single Azure Cosmos DB account to use with the DocumentDB API.
Scale Azure Cosmos DB	
Replicate Azure Cosmos DB account in multiple regions and configure failover priorities	Globally replicates account data into multiple regions with a specified failover priority.
Secure Azure Cosmos DB	
Get account keys	Gets the primary and secondary master write keys and primary and secondary read-only keys for the account.
Get MongoDB connection string	Gets the connection string to connect your MongoDB app to your Azure Cosmos DB account.
Regenerate account keys	Regenerates the master or read-only key for the account.
Create a firewall	Creates an inbound IP access control policy to limit access to the account from an approved set of machines and/or cloud services.
High availability, disaster recovery, backup and restore	
Configure failover policy	Sets the failover priority of each region in which the account is replicated.

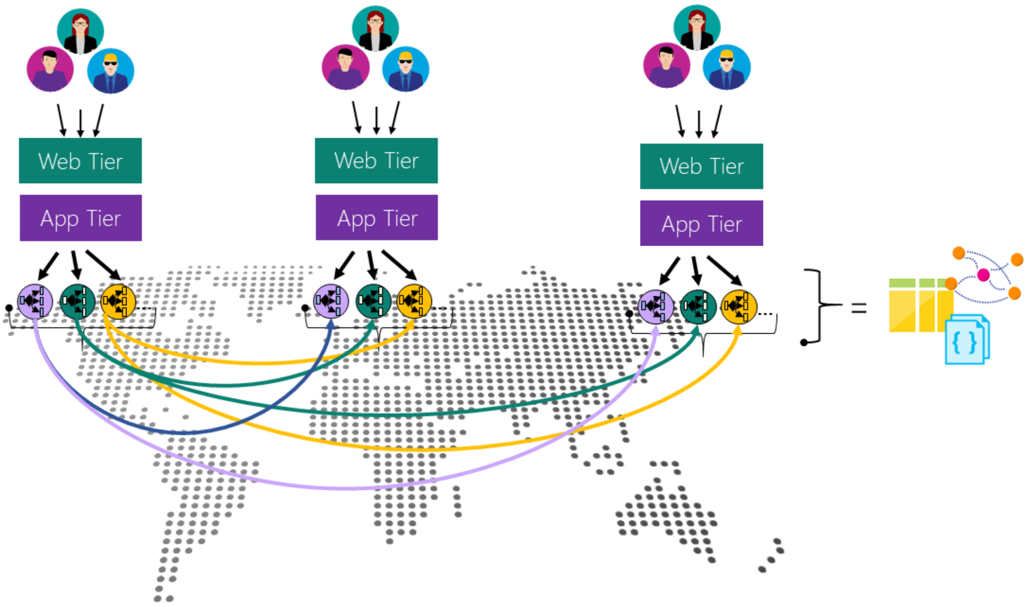
How to distribute data globally with Azure Cosmos DB?

6/9/2017 • 16 min to read • Edit Online

Azure is ubiquitous - it has a global footprint across 30+ geographical regions and is continuously expanding. With its worldwide presence, one of the differentiated capabilities Azure offers to its developers is the ability to build, deploy, and manage globally distributed applications easily.

Azure Cosmos DB is Microsoft's globally distributed, multi-model database service for mission-critical applications. Azure Cosmos DB provides turn-key global distribution, elastic scaling of throughput and storage worldwide, single-digit millisecond latencies at the 99th percentile, five well-defined consistency levels, and guaranteed high availability, all backed by industry-leading SLAs. Azure Cosmos DB automatically indexes data without requiring you to deal with schema and index management. It is multi-model and supports document, key-value, graph, and columnar data models. As a cloud-born service, Azure Cosmos DB is carefully engineered with multi-tenancy and global distribution from the ground up.

A single Azure Cosmos DB collection partitioned and distributed across three Azure regions



As we have learned while building Azure Cosmos DB, adding global distribution cannot be an afterthought - it cannot be "bolted-on" atop a "single site" database system. The capabilities offered by a globally distributed database span beyond that of traditional geographical disaster recovery (Geo-DR) offered by "single-site" databases. Single site databases offering Geo-DR capability are a strict subset of globally distributed databases.

With Azure Cosmos DB's turnkey global distribution, developers do not have to build their own replication scaffolding by employing either the Lambda pattern (for example, AWS DynamoDB replication) over the database log or by doing "double writes" across multiple regions. We do not recommend these approaches since it is impossible to ensure correctness of such approaches and provide sound SLAs.

In this article, we provide an overview of Azure Cosmos DB's global distribution capabilities. We also describe Azure Cosmos DB's unique approach to providing comprehensive SLAs.

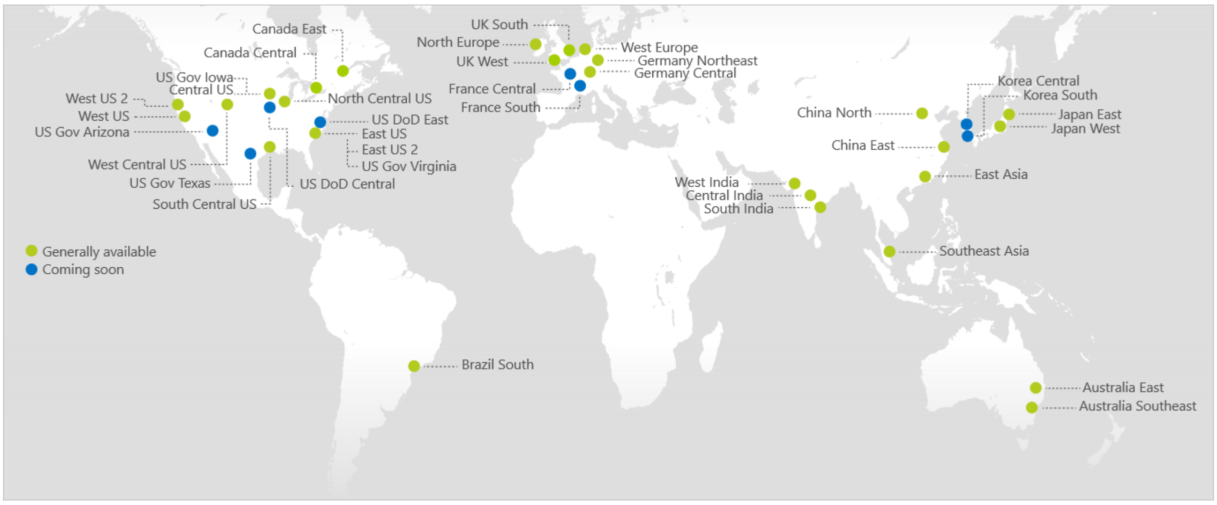
Enabling turn-key global distribution

Azure Cosmos DB provides the following capabilities to enable you to easily write planet scale applications. These capabilities are available via the Azure Cosmos DB's resource provider-based REST APIs as well as the Azure portal.

Ubiquitous regional presence

Azure is constantly growing its geographical presence by bringing new regions online. Azure Cosmos DB is available in all new Azure regions by default. This allows you to associate a geographical region with your Azure Cosmos DB database account as soon as Azure opens the new region for business.

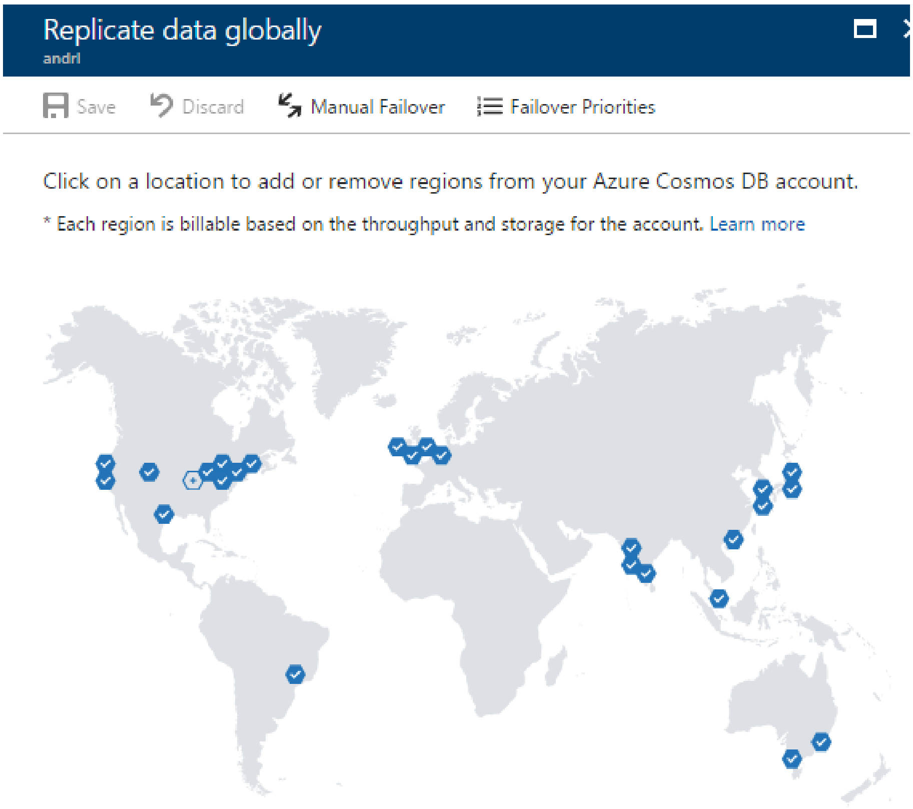
Azure Cosmos DB is available in all Azure regions by default



Associating an unlimited number of regions with your Azure Cosmos DB database account

Azure Cosmos DB allows you to associate any number of Azure regions with your Azure Cosmos DB database account. Outside of geo-fencing restrictions (for example, China, Germany), there are no limitations on the number of regions that can be associated with your Azure Cosmos DB database account. The following figure shows a database account configured to span across 25 Azure regions.

A tenant's Azure Cosmos DB database account spanning 25 Azure regions



Click on a location to add or remove regions from your Azure Cosmos DB account.

* Each region is billable based on the throughput and storage for the account. Learn more

Policy-based geo-fencing

Azure Cosmos DB is designed to have policy-based geo-fencing capabilities. Geo-fencing is an important component to ensure data governance and compliance restrictions and may prevent associating a specific region with your account. Examples of geo-fencing include (but are not restricted to), scoping global distribution to the regions within a sovereign cloud (for example, China and Germany), or within a government taxation boundary (for example, Australia). The policies are controlled using the metadata of your Azure subscription.

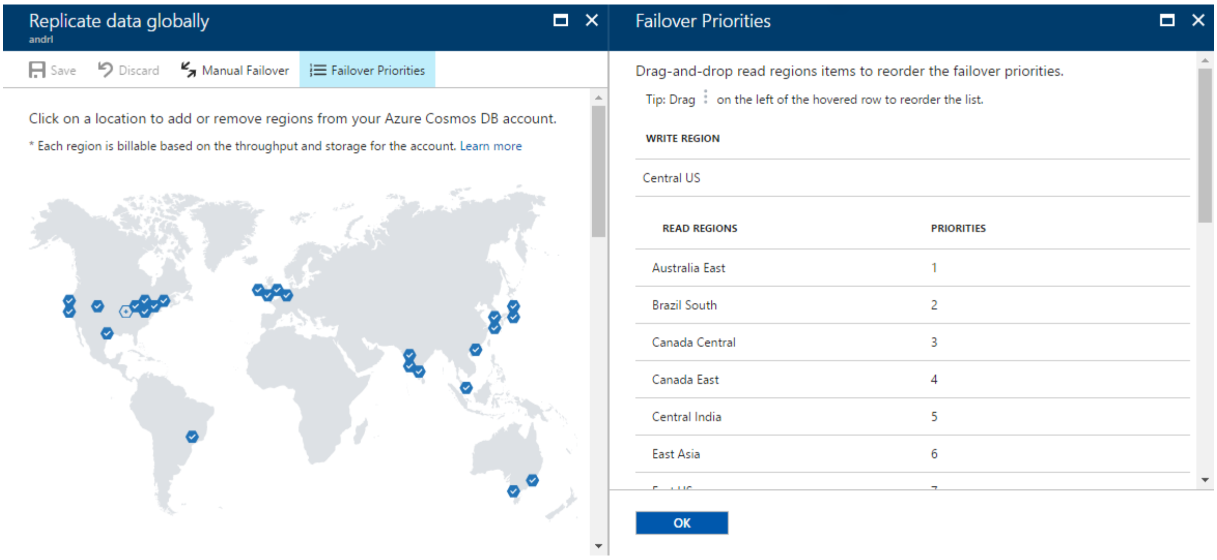
Dynamically add and remove regions

Azure Cosmos DB allows you to add (associate) or remove (dissociate) regions to your database account at any point in time (see preceding figure). By virtue of replicating data across partitions in parallel, Azure Cosmos DB ensures that when a new region comes online, Azure Cosmos DB is available within 30 minutes anywhere in the world for up to 100 TBs.

Failover priorities

To control exact sequence of regional failovers when there is a multi-regional outage, Azure Cosmos DB enables you to associate the priority to various regions associated with the database account (see the following figure). Azure Cosmos DB ensures that the automatic failover sequence occurs in the priority order you specified. For more information about regional failovers, see Automatic regional failovers for business continuity in Azure Cosmos DB.

A tenant of Azure Cosmos DB can configure the failover priority order (right pane) for regions associated with a database account



Click on a location to add or remove regions from your Azure Cosmos DB account.

* Each region is billable based on the throughput and storage for the account. Learn more

Drag-and-drop read regions items to reorder the failover priorities.

Tip: Drag ⋮ on the left of the hovered row to reorder the list.

WRITE REGION
Central US

READ REGIONS	PRIORITIES
Australia East	1
Brazil South	2
Canada Central	3
Canada East	4
Central India	5
East Asia	6

Dynamically taking a region "offline"

Azure Cosmos DB enables you to take your database account offline in a specific region and bring it back online later. Regions marked offline do not actively participate in replication and are not part of the failover sequence. This enables you to freeze the last known good database image in one of the read regions before rolling out potentially risky upgrades to your application.

Multiple, well-defined consistency models for globally replicated databases

Azure Cosmos DB exposes multiple well-defined consistency levels backed by SLAs. You can choose a specific consistency model (from the available list of options) depending on the workload/scenarios.

Tunable consistency for globally replicated databases

Azure Cosmos DB allows you to programmatically override and relax the default consistency choice on a per request basis, at runtime.

Dynamically configurable read and write regions

Azure Cosmos DB enables you to configure the regions (associated with the database) for "read", "write" or "read/write" regions.

Elastically scaling throughput across Azure regions

You can elastically scale an Azure Cosmos DB collection by provisioning throughput programmatically. The throughput is applied to all the regions the collection is distributed in.

Geo-local reads and writes

The key benefit of a globally distributed database is to offer low latency access to the data anywhere in the world. Azure Cosmos DB offers low latency guarantees at P99 for various database operations. It ensures that all reads are routed to the closest local read region. To serve a read request, the quorum local to the region in which the read is issued is used; the same applies to the writes. A write is acknowledged only after a majority of replicas has durably committed the write locally but without being gated on remote replicas to acknowledge the writes. Put differently, the replication protocol of Azure Cosmos DB operates under the assumption that the read and write quorums are always local to the read and write regions, respectively, in which the request is issued.

Manually initiate regional failover

Azure Cosmos DB allows you to trigger the failover of the database account to validate the *end to end* availability properties of the entire application (beyond the database). Since both the safety and liveness properties of the failure detection and leader election are guaranteed, Azure Cosmos DB guarantees *zero-data-loss* for a tenant-initiated manual failover operation.

Automatic failover

Azure Cosmos DB supports automatic failover in case of one or more regional outages. During a regional failover, Azure Cosmos DB maintains its read latency, uptime availability, consistency, and throughput SLAs. Azure Cosmos DB provides an upper bound on the duration of an automatic failover operation to complete. This is the window of potential data loss during the regional outage.

Designed for different failover granularities

Currently the automatic and manual failover capabilities are exposed at the granularity of the database account. Note, internally Azure Cosmos DB is designed to offer *automatic* failover at finer granularity of a database, collection, or even a partition (of a collection owning a range of keys).

Multi-homing APIs in Azure Cosmos DB

Azure Cosmos DB allows you to interact with the database using either logical (region agnostic) or physical (region-specific) endpoints. Using logical endpoints ensures that the application can transparently be multi-homed in case of failover. The latter, physical endpoints, provide fine-grained control to the application to redirect reads and writes to specific regions.

You can find information on how to configure read preferences for the DocumentDB API, Graph API, Table API, and MongoDB API in their respective linked articles.

Transparent and consistent database schema and index migration

Azure Cosmos DB is fully schema agnostic. The unique design of its database engine allows it to automatically and synchronously index all of the data it ingests without requiring any schema or secondary indices from you. This enables you to iterate your globally distributed application rapidly without worrying about database schema and index migration or coordinating multi-phase application rollouts of schema changes. Azure Cosmos DB guarantees that any changes to indexing policies explicitly made by you does not result into degradation of either performance or availability.

Comprehensive SLAs (beyond just high availability)

As a globally distributed database service, Azure Cosmos DB offers well-defined SLA for **data-loss**, **availability**, **latency at P99**, **throughput** and **consistency** for the database as a whole, regardless of the number of regions associated with the database.

Latency guarantees

The key benefit of a globally distributed database service like Azure Cosmos DB is to offer low latency access to your data anywhere in the world. Azure Cosmos DB offers guaranteed low latency at P99 for various database operations. The replication protocol that Azure Cosmos DB employs ensures that the database operations (ideally, both reads and writes) are always performed in the region local to that of the client. The latency SLA of Azure Cosmos DB includes P99 for both reads, (synchronously) indexed writes and queries for various request and response sizes. The latency guarantees for writes include durable majority quorum commits within the local datacenter.

Latency's relationship with consistency

For a globally distributed service to offer strong consistency in a globally distributed setup, it needs to synchronously replicate the writes or synchronous perform cross-region reads – the speed of light and the wide area network reliability dictate that strong consistency results in high latencies and low availability of database operations. Hence, in order to offer guaranteed low latencies at P99 and 99.99 availability, the service must employ asynchronous replication. This in-turn requires that the service must also offer well-defined, relaxed consistency choice(s) – weaker than strong (to offer low latency and availability guarantees) and ideally stronger than "eventual" consistency (to offer an intuitive programming model).

Azure Cosmos DB ensures that a read operation is not required to contact replicas across multiple regions to deliver the specific consistency level guarantee. Likewise, it ensures that a write operation does not get blocked while the data is being replicated across all the regions (i.e. writes are asynchronously replicated across regions). For multi-region database accounts multiple relaxed consistency levels are available.

Latency's relationship with availability

Latency and availability are the two sides of the same coin. We talk about latency of the operation in steady state and availability, in the face of failures. From the application standpoint, a slow running database operation is indistinguishable from a database that is unavailable.

To distinguish high latency from unavailability, Azure Cosmos DB provides an absolute upper bound on latency of various database operations. If the database operation takes longer than the upper bound to complete, Azure Cosmos DB returns a timeout error. The Azure Cosmos DB availability SLA ensures that the timeouts are counted against the availability SLA.

Latency's relationship with throughput

Azure Cosmos DB does not make you choose between latency and throughput. It honors the SLA for both latency at P99 and deliver the throughput that you have provisioned.

Consistency guarantees

While the strong consistency model is the gold standard of programmability, it comes at the steep price of high latency (in steady state) and loss of availability (in the face of failures).

Azure Cosmos DB offers a well-defined programming model to you to reason about replicated data's consistency. In order to enable you to build multi-homed applications, the consistency models exposed by Azure Cosmos DB are designed to be region-agnostic and not depend on the region from where the reads and writes are served.

Azure Cosmos DB's consistency SLA guarantees that 100% of read requests will meet the consistency guarantee for the consistency level requested by you (either the default consistency level on the database account or the overridden value on the request). A read request is considered to have met the consistency SLA if all the consistency guarantees associated with the consistency level are satisfied. The following table captures the consistency guarantees that correspond to specific consistency levels offered by Azure Cosmos DB.

Consistency guarantees associated with a given consistency level in Azure Cosmos DB

Consistency Level	Consistency Characteristics	SLA
Session	Read your own write	100%
	Monotonic read	100%
	Consistent prefix	100%
Bounded staleness	Monotonic read (within a region)	100%
	Consistent prefix	100%
	Staleness bound < K,T	100%
Consistent prefix	Consistent prefix	100%
Strong	Linearizable	100%

Consistency's relationship with availability

The impossibility result of the CAP theorem proves that it is indeed impossible for the system to remain available and offer linearizable consistency in the face of failures. The database service must choose to be either CP or AP - CP systems forgo availability in favor of linearizable consistency while the AP systems forgo linearizable consistency in favor of availability. Azure Cosmos DB never violates the requested consistency level, which formally makes it a CP system. However, in practice consistency is not an all or nothing proposition – there are multiple well-defined consistency models along the consistency spectrum between linearizable and eventual consistency. In Azure Cosmos DB, we have tried to identify several of the relaxed consistency models with real world applicability and an intuitive programming model. Azure Cosmos DB navigates the consistency-availability tradeoffs by offering 99.99 availability SLA along with multiple relaxed yet well-defined consistency levels.

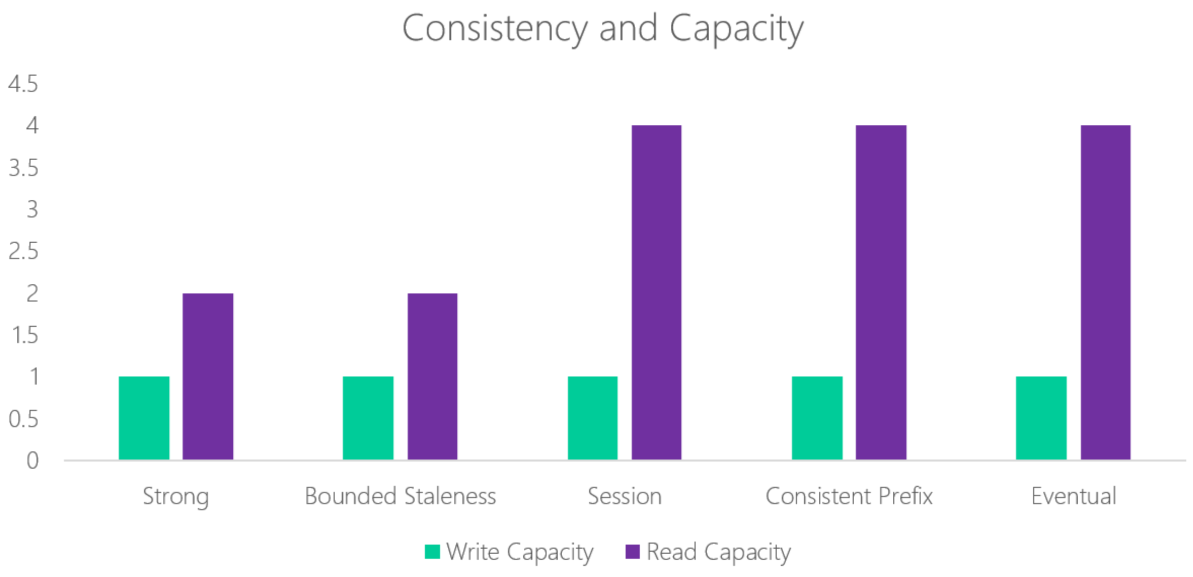
Consistency's relationship with latency

A more comprehensive variation of CAP was proposed by Prof. Daniel Abadi and it is called PACELC, which also accounts for latency and consistency tradeoffs in steady state. It states that in steady state, the database system must choose between consistency and latency. With multiple relaxed consistency models (backed by asynchronous replication and local read, write quorums), Azure Cosmos DB ensures that all reads and writes are local to the read and write regions respectively. This allows Azure Cosmos DB to offer low latency guarantees within the region for the consistency levels.

Consistency's relationship with throughput

Since the implementation of a specific consistency model depends on the choice of a quorum type, the throughput also varies based on the choice of consistency. For instance, in Azure Cosmos DB, the throughput with strongly consistent reads is roughly half to that of eventually consistent reads.

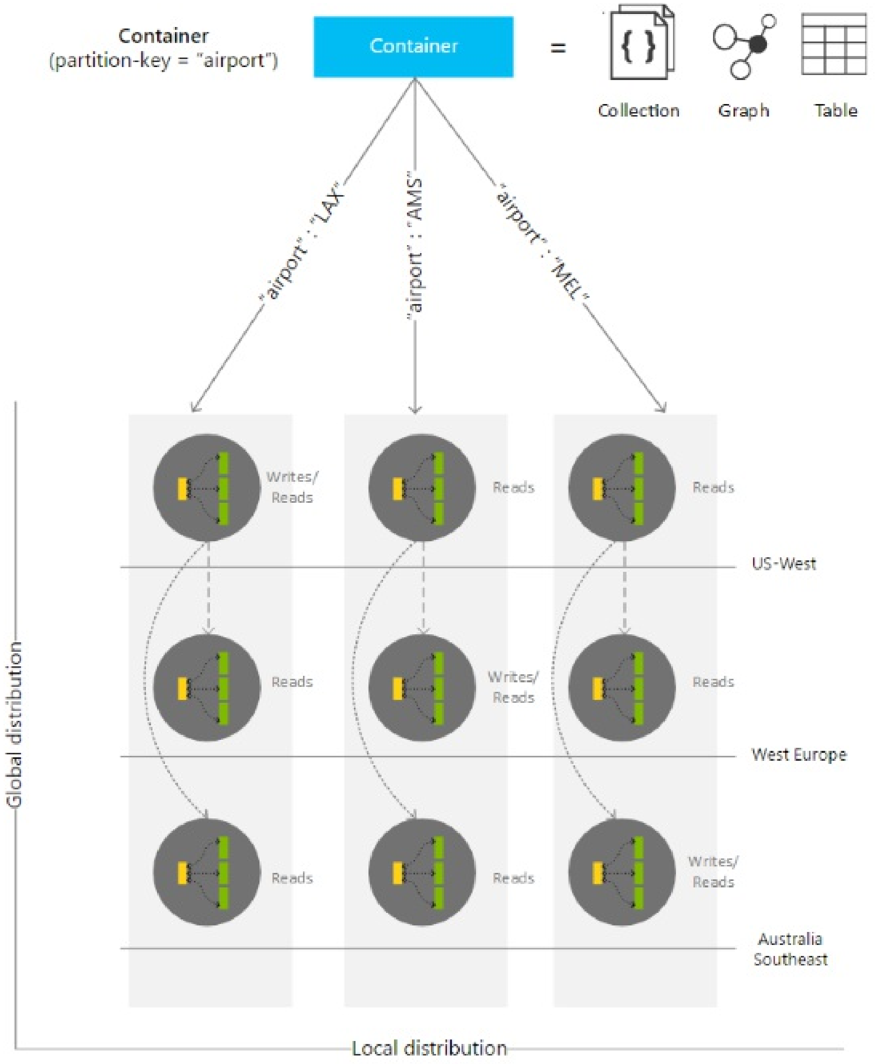
Relationship of read capacity for a specific consistency level in Azure Cosmos DB



Throughput guarantees

Azure Cosmos DB allows you to scale throughput (as well as, storage), elastically across different regions depending on the demand.

A single Azure Cosmos DB collection partitioned (across three shards) and then distributed across three Azure regions

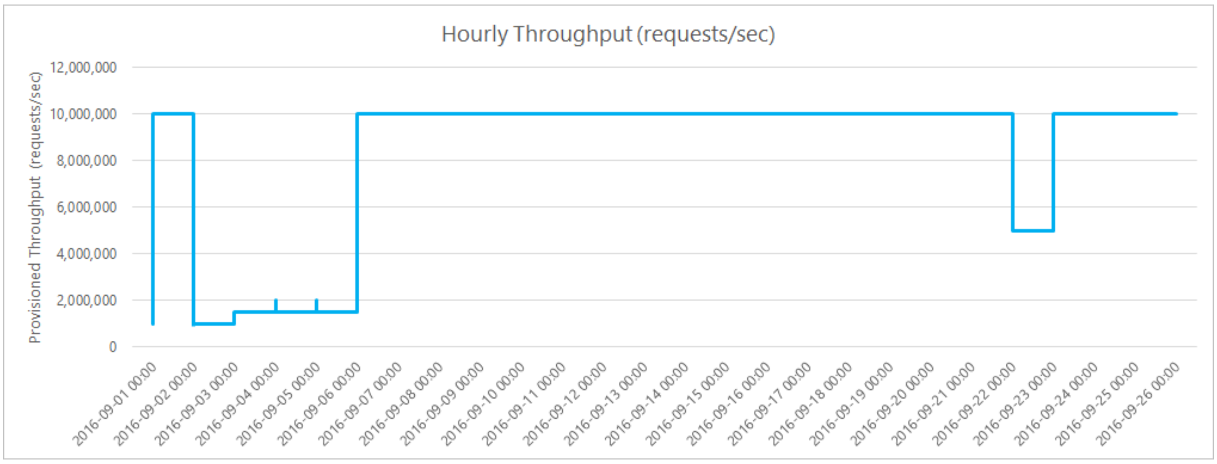


An Azure Cosmos DB collection gets distributed using two dimensions – within a region and then across regions. Here's how:

- Within a single region, an Azure Cosmos DB collection is scaled out in terms of resource partitions. Each resource partition manages a set of keys and is strongly consistent and highly available by virtue of state machine replication among a set of replicas. Azure Cosmos DB is a fully resource governed system where a resource partition is responsible for delivering its share of throughput for the budget of system resources allocated to it. The scaling of an Azure Cosmos DB collection is completely transparent – Azure Cosmos DB manages the resource partitions and splits and merges it as needed.
- Each of the resource partitions is then distributed across multiple regions. Resource partitions owning the same set of keys across various regions form partition set (see preceding figure). Resource partitions within a partition set are coordinated using state machine replication across the multiple regions. Depending on the consistency level configured, the resource partitions within a partition set are configured dynamically using different topologies (for example, star, daisy-chain, tree etc.).

By virtue of a highly responsive partition management, load balancing and strict resource governance, Azure Cosmos DB allows you to elastically scale throughput across multiple Azure regions on an Azure Cosmos DB collection. Changing throughput on a collection is a runtime operation in Azure Cosmos DB - like with other database operations Azure Cosmos DB guarantees the absolute upper bound on latency for your request to change the throughput. As an example, the following figure shows a customer's collection with elastically provisioned throughput (ranging from 1M-10M requests/sec across two regions) based on the demand.

A customer's collection with elastically provisioned throughput (1M-10M requests/sec)



Throughput's relationship with consistency

Same as Consistency's relationship with throughput.

Throughput's relationship with availability

Azure Cosmos DB continues to maintain its availability when the changes are made to the throughput. Azure Cosmos DB transparently manages partitions (for example, split, merge, clone operations) and ensures that the operations do not degrade performance or availability, while the application elastically increases or decreases throughput.

Availability guarantees

Azure Cosmos DB offers a 99.99% uptime availability SLA for each of the data and control plane operations. As described earlier, Azure Cosmos DB's availability guarantees include an absolute upper bound on latency for every data and control plane operations. The availability guarantees are steadfast and do not change with the number of regions or geographical distance between regions. Availability guarantees apply with both manual as well as, automatic failover. Azure Cosmos DB offers transparent multi-homing APIs that ensure that your application can operate against logical endpoints and can transparently route the requests to the new region in case of failover. Put differently, your application does not need to be redeployed upon regional failover and the availability SLAs are maintained.

Availability's relationship with consistency, latency, and throughput

Availability’s relationship with consistency, latency, and throughput is described in Consistency's relationship with availability, Latency's relationship with availability and Throughput's relationship with availability.

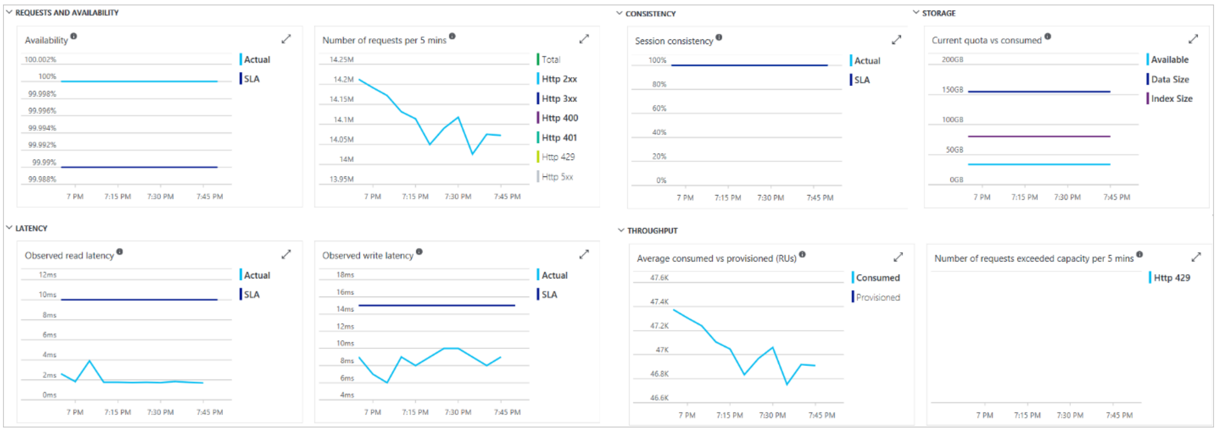
Guarantees and system behavior for "data loss"

In Azure Cosmos DB, each partition (of a collection) is made highly available by a number of replicas, which are spread across at least 10-20 fault domains. All writes are synchronously and durably committed by a majority quorum of replicas before they are acknowledged to the client. Asynchronous replication is applied with coordination across partitions spread across multiple regions. Azure Cosmos DB guarantees that there is no data loss for a tenant-initiated manual failover. During automatic failover, Azure Cosmos DB guarantees an upper bound of the configured bounded staleness interval on the data loss window as part of its SLA.

Customer facing SLA Metrics

Azure Cosmos DB transparently exposes the throughput, latency, consistency and availability metrics. These metrics are accessible programmatically and via the Azure portal (see following figure). You can also set up alerts on various thresholds using Azure Application Insights.

Consistency, Latency, Throughput, and Availability metrics are transparently available to each tenant



Next Steps

- To implement global replication on your Azure Cosmos DB account using the Azure portal, see How to perform Azure Cosmos DB global database replication using the Azure portal.
- To learn about how to implement multi-master architectures with Azure Cosmos DB, see Multi-master database architectures with Azure Cosmos DB.
- To learn more about how automatic and manual failovers work in Azure Cosmos DB, see Regional Failovers in Azure Cosmos DB.

References

1. Eric Brewer. Towards Robust Distributed Systems
2. Eric Brewer. CAP Twelve Years Later – How the rules have changed
3. Gilbert, Lynch. - Brewer's Conjecture and Feasibility of Consistent, Available, Partition Tolerant Web Services
4. Daniel Abadi. Consistency Tradeoffs in Modern Distributed Database Systems Design
5. Martin Kleppmann. Please stop calling databases CP or AP
6. Peter Bailis et al. Probabilistic Bounded Staleness (PBS) for Practical Partial Quorums
7. Naor and Wool. Load, Capacity and Availability in Quorum Systems
8. Herlihy and Wing. Lineralizability: A correctness condition for concurrent objects
9. Azure Cosmos DB SLA

How to partition and scale in Azure Cosmos DB

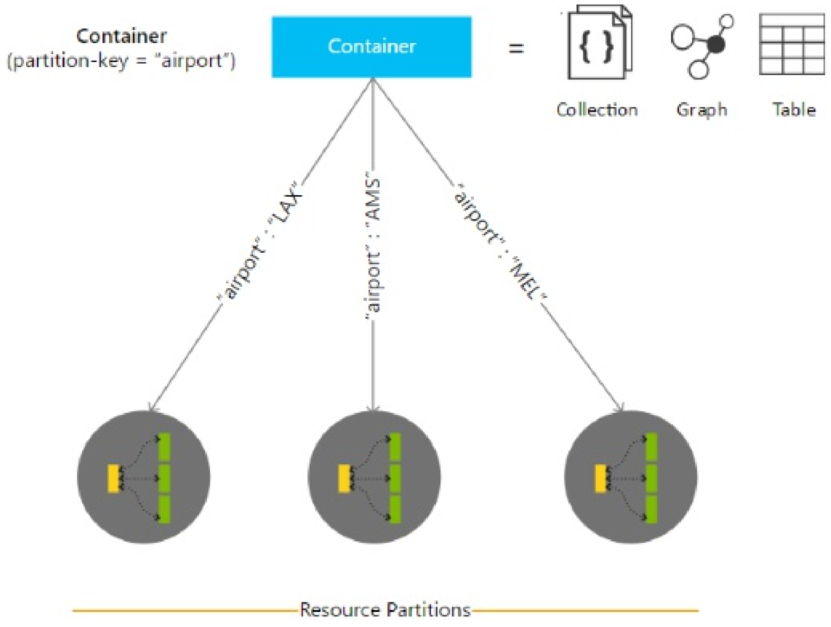
6/13/2017 • 10 min to read • Edit Online

Microsoft Azure Cosmos DB is a global distributed, multi-model database service 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 for all the data models in Azure Cosmos DB, and describes how you can configure Azure Cosmos DB containers to effectively scale your applications.

Partitioning and partition keys are also covered in this Azure Friday video with Scott Hanselman and Azure Cosmos DB Principal Engineering Manager, Shireesh Thota.

Partitioning in Azure Cosmos DB

In Azure Cosmos DB, you can store and query schema-less data with order-of-millisecond response times at any scale. Cosmos DB provides containers for storing data called **collections (for document), graphs, or tables**. Containers are logical resources and can span one or more physical partitions or servers. The number of partitions is determined by Cosmos DB based on the storage size and the provisioned throughput of the container. Every partition in Cosmos DB has a fixed amount of SSD-backed storage associated with it, and is replicated for high availability. Partition management is fully managed by Azure Cosmos DB, and you do not have to write complex code or manage your partitions. Cosmos DB containers are unlimited in terms of storage and throughput.



Partitioning is transparent to your application. Cosmos DB supports fast reads and writes, queries, transactional logic, consistency levels, and fine-grained access control via methods/APIs to a single container resource. The service handles distributing data across partitions and routing query requests to the right partition.

How does partitioning work? Each item must have a partition key and a row key, which uniquely identify it. Your partition key acts as a logical partition for your data, and provides Cosmos DB with a natural boundary for distributing data across partitions. In brief, here is how partitioning works in Azure Cosmos DB:

- You provision a Cosmos DB container with `T` requests/s throughput
- Behind the scenes, Cosmos DB provisions partitions needed to serve `T` requests/s. If `T` is higher than the maximum throughput per partition `t`, then Cosmos DB provisions `N` = `T/t` partitions
- Cosmos DB allocates the key space of partition key hashes evenly across the `N` partitions. So, each partition (physical partition) hosts 1-N partition key values (logical partitions)
- When a physical partition `p` reaches its storage limit, Cosmos DB seamlessly splits `p` into two new partitions `p1` and `p2` and distributes values corresponding to roughly half the keys to each of the partitions. This split operation is invisible to your application.
- Similarly, when you provision throughput higher than `t*N` throughput, Cosmos DB splits one or more of your partitions to support the higher throughput

The semantics for partition keys are slightly different to match the semantics of each API, as shown in the following table:

API	PARTITION KEY	ROW KEY
DocumentDB	custom partition key path	fixed `id`
MongoDB	custom shard key	fixed `_id`
Graph	custom partition key property	fixed `id`
Table	fixed `PartitionKey`	fixed `RowKey`

Cosmos DB uses hash-based partitioning. When you write an item, Cosmos DB hashes the partition key value and use the hashed result to determine which partition to store the item in. Cosmos DB stores all items with the same partition key in the same physical partition. The choice of the partition key is an important decision that you have to make at design time. You must pick a property name that has a wide range of values and has even access patterns.

NOTE

It is a best practice to have a partition key with many distinct values (100s-1000s at a minimum).

Azure Cosmos DB containers can be created as "fixed" or "unlimited." Fixed-size containers have a maximum limit of 10 GB and 10,000 RU/s throughput. Some APIs allow the partition key to be omitted for fixed-size containers. To create a container as unlimited, you must specify a minimum throughput of 2500 RU/s.

Partitioning and provisioned throughput

Cosmos DB is designed for predictable performance. When you create a container, you reserve throughput in terms of **request units (RU) per second with a potential add-on for RU per minute**. Each request is assigned a request unit charge that is proportionate to the amount of system resources like CPU, Memory, and IO consumed by the operation. A read of a 1-KB document with Session consistency consumes one request unit. A read is 1 RU regardless of the number of items stored or the number of concurrent requests running at the same time. Larger items 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.

> NOTE
>
> To achieve the full throughput of the container, you must choose a partition key that allows you to evenly distribute requests among some distinct partition key values.

Working with the Azure Cosmos DB APIs

You can use the Azure portal or Azure CLI to create containers and scale them at any time. This section shows how to create containers and specify the throughput and partition key definition in each of the supported APIs.

DocumentDB API

The following sample shows how to create a container (collection) using the DocumentDB API. You can find more details in Partitioning with DocumentDB API.

```
DocumentClient client = new DocumentClient(new Uri(endpoint), authKey);
await client.CreateDatabaseAsync(new Database { Id = "db" });

DocumentCollection myCollection = new DocumentCollection();
myCollection.Id = "coll";
myCollection.PartitionKey.Paths.Add("/deviceId");

await client.CreateDocumentCollectionAsync(
    UriFactory.CreateDatabaseUri("db"),
    myCollection,
    new RequestOptions { OfferThroughput = 20000 });
```

You can read an item (document) using the `GET` method in the REST API or using `ReadDocumentAsync` in one of the SDKs.

```
// Read document. Needs the partition key and the ID to be specified
DeviceReading document = await client.ReadDocumentAsync<DeviceReading>(
  UriFactory.CreateDocumentUri("db", "coll", "XMS-001-FE24C"),
  new RequestOptions { PartitionKey = new PartitionKey("XMS-0001") });
```

MongoDB API

With the MongoDB API, you can create a sharded collection through your favorite tool, driver, or SDK. In this example, we use the Mongo Shell for the collection creation.

In the Mongo Shell:

```
db.runCommand( { shardCollection: "admin.people", key: { region: "hashed" } } )
```

Results:

```
{
    "_t" : "ShardCollectionResponse",
    "ok" : 1,
    "collectionsharded" : "admin.people"
}
```

Table API

With the Table API, you specify the throughput for tables in the appSettings configuration for your application:

```
<configuration>
  <appSettings>
    <!--Table creation options -->
    <add key="TableThroughput" value="700"/>
  </appSettings>
</configuration>
```

Then you create a table using the Azure Table storage SDK. The partition key is implicitly created as the `PartitionKey` value.

```
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();

CloudTable table = tableClient.GetTableReference("people");
table.CreateIfNotExists();
```

You can retrieve a single entity using the following snippet:

```
// Create a retrieve operation that takes a customer entity.
TableOperation retrieveOperation = TableOperation.Retrieve<CustomerEntity>("Smith", "Ben");

// Execute the retrieve operation.
TableResult retrievedResult = table.Execute(retrieveOperation);
```

See Developing with the Table API for more details.

Graph API

With the Graph API, you must use the Azure portal or CLI to create containers. Alternatively, since Azure Cosmos DB is multi-model, you can use one of the other models to create and scale your graph container.

You can read any vertex or edge using the partition key and id in Gremlin. For example, for a graph with region ("USA") as the partition key, and "Seattle" as the row key, you can find a vertex using the following syntax:

```
g.V(['USA', 'Seattle'])
```

Same with edges, you can reference an edge using the partition key and row key.

```
g.E(['USA', 'I5'])
```

See Gremlin support for Cosmos DB for more details.

Designing for partitioning

To scale effectively with Azure Cosmos DB, you need to pick a good partition key when you create your container. There are two key considerations for choosing a partition key:

- **Boundary for query and transactions**: 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 items, but this may limit the scalability of your solution. At the other extreme, you could assign a unique partition key for each item, 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.

- **No storage and performance bottlenecks**: It is important to pick a property that allows writes to be distributed across various distinct values. Requests to the same partition key cannot exceed the throughput of a single partition, and are throttled. So it is important to pick a partition key that does not result in "hot spots" within your application. Since all the data for a single partition key must be stored within a partition, it is also recommended to avoid partition keys that have high volumes of data for the same value.

Let's look at a few real-world scenarios, and good partition keys for each:

- 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 for example, 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.

In some use cases like IoT and user profiles, 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 common use cases of Cosmos DB 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 depends 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 rollup of the timestamp, for example, 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 written heavy, which is more common, you should use a partition key that's not based on timestamp so that Cosmos DB can distribute writes evenly across various 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 containers, one for each day/month and the partition key is a granular property like hostname. This has the benefit that you can set different throughput based on the time window, for example, the container 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 Cosmos DB, there are two popular patterns – one partition key per tenant, and one container per tenant. Here are the pros and cons for each:

- One Partition Key per tenant: In this model, tenants are collocated within a single container. But queries and inserts for items within a single tenant can be performed against a single partition. You can also implement transactional logic across all items within a tenant. Since multiple tenants share a container, you can save storage and throughput costs by pooling resources for tenants within a single container 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 container vs targeted increases for tenants.
- One Container per tenant: Each tenant has its own container. In this model, you can reserve performance per tenant. With Cosmos DB's new provisioning pricing model, this model is more cost-effective for multi-tenant applications with a few tenants.

You can also use a combination/tiered approach that collocates small tenants and migrates larger tenants to their own container.

Next steps

In this article, we provided an overview for an overview of concepts and best practices for partitioning with any Azure Cosmos DB API.

- Learn about provisioned throughput in Azure Cosmos DB
- Learn about global distribution in Azure Cosmos DB

Tunable data consistency levels in Azure Cosmos DB

5/30/2017 • 8 min to read • Edit Online

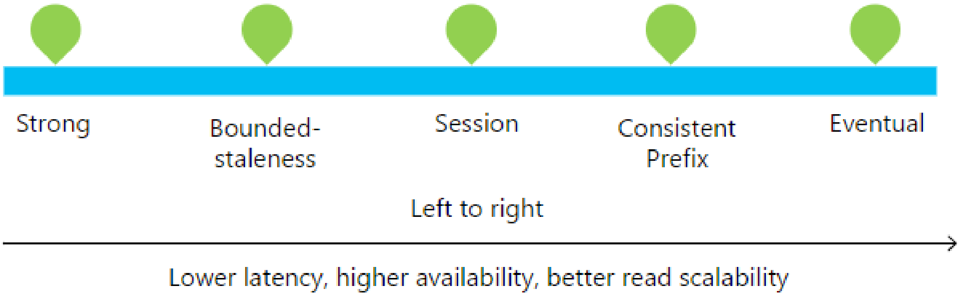
Azure Cosmos DB is designed from the ground up with global distribution in mind for every data model. It is designed to offer predictable low latency guarantees, a 99.99% availability SLA, and multiple well-defined relaxed consistency models. Currently, Azure Cosmos DB provides five consistency levels: strong, bounded-staleness, session, and eventual.

Besides **strong** and **eventual consistency** models commonly offered by distributed databases, Azure Cosmos DB offers three more carefully codified and operationalized consistency models, and has validated their usefulness against real world use cases. These are the **bounded staleness**, **session**, and **consistent prefix** consistency levels. Collectively these five consistency levels enable you to make well-reasoned trade-offs between consistency, availability, and latency.

Distributed databases and consistency

Commercial distributed databases fall into two categories: databases that do not offer well-defined, provable consistency choices at all, and databases which offer two extreme programmability choices (strong vs. eventual consistency).

The former burdens application developers with minutia of their replication protocols and expects them to make difficult tradeoffs between consistency, availability, latency, and throughput. The latter puts a pressure to choose one of the two extremes. Despite the abundance of research and proposals for more than 50 consistency models, the distributed database community has not been able to commercialize consistency levels beyond strong and eventual consistency. Cosmos DB allows developers to choose between five well-defined consistency models along the consistency spectrum – strong, bounded staleness, session, consistent prefix, and eventual.



The following table illustrates the specific guarantees each consistency level provides.

Consistency Levels and guarantees

CONSISTENCY LEVEL	GUARANTEES
Strong	Linearizability
Bounded Staleness	Consistent Prefix. Reads lag behind writes by k prefixes or t interval

CONSISTENCY LEVEL	GUARANTEES
Session	Consistent Prefix. Monotonic reads, monotonic writes, read-your-writes, write-follows-reads
Consistent Prefix	Updates returned are some prefix of all the updates, with no gaps
Eventual	Out of order reads

You can configure the default consistency level on your Cosmos DB account (and later override the consistency on a specific read request). Internally, the default consistency level applies to data within the partition sets which may be span regions. About 73% of our tenants use session consistency and 20% prefer bounded staleness. We observe that approximately 3% of our customers experiment with various consistency levels initially before settling on a specific consistency choice for their application. We also observe that only 2% of our tenants override consistency levels on a per request basis.

In Cosmos DB, reads served at session, consistent prefix and eventual consistency are twice as cheap as reads with strong or bounded staleness consistency. Cosmos DB has industry leading comprehensive 99.99% SLAs including consistency guarantees along with availability, throughput, and latency. We employ a linearizability checker, which continuously operates over our service telemetry and openly reports any consistency violations to you. For bounded staleness, we monitor and report any violations to k and t bounds. For all five relaxed consistency levels, we also report the probabilistic bounded staleness metric directly to you.

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. As with 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 collections (and databases) under your Cosmos DB account. By default, all reads and queries issued against the user-defined resources use the default consistency level specified on the database account. You can relax the consistency level of a specific read/query request using in each of the supported APIs. There are five types of consistency levels supported by the Azure Cosmos DB replication protocol that provide a clear trade-off between specific consistency guarantees and performance, as described in this section.

Strong:

- Strong consistency offers a linearizability guarantee with the reads guaranteed to return the most recent version of an item.
- 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.
- Azure Cosmos DB accounts that are configured to use strong consistency cannot associate more than one Azure region with their Azure Cosmos DB 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 an item or *t* time-interval.
- Therefore, when choosing bounded staleness, the "staleness" can be configured in two ways: number of versions *K* of the item by which the reads lag behind the writes, and the time interval *t*
- Bounded staleness offers total global order except within the "staleness window." 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.
- Azure Cosmos DB accounts that are configured with bounded staleness consistency can associate any number of Azure regions with their Azure Cosmos DB 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.
- Azure Cosmos DB accounts that are configured with session consistency can associate any number of Azure regions with their Azure Cosmos DB 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

Consistent Prefix:

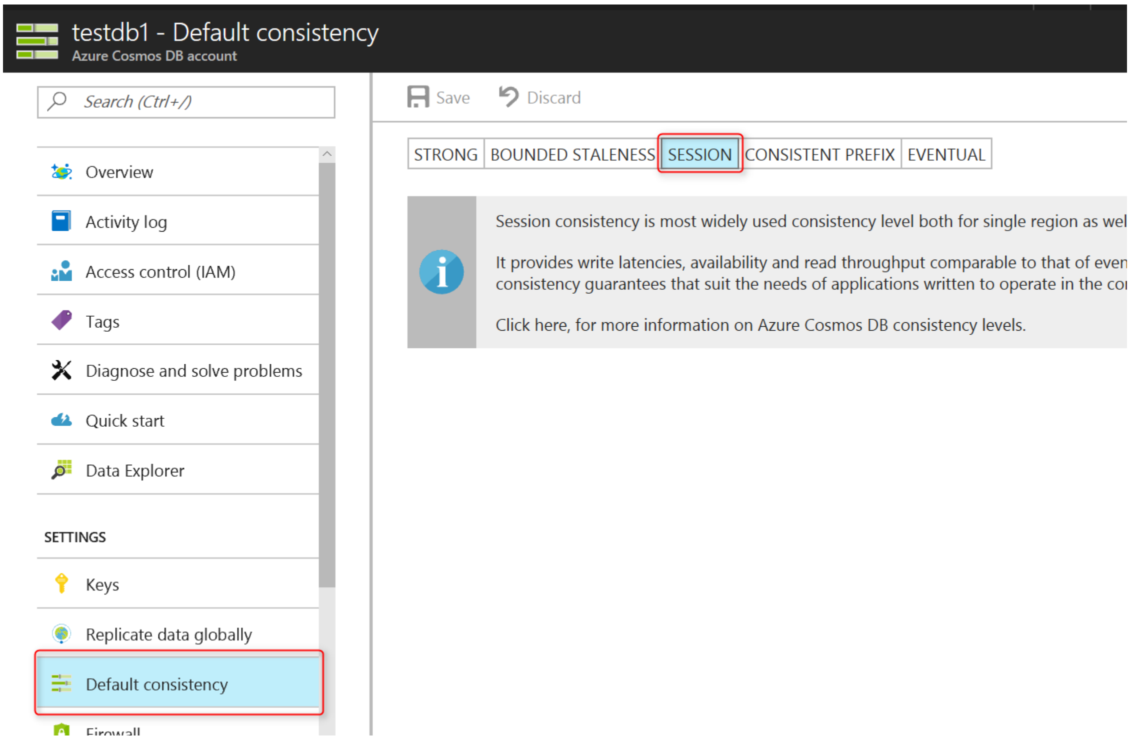
- Consistent prefix guarantees that in absence of any further writes, the replicas within the group eventually converge.
- Consistent prefix guarantees that reads never see out of order writes. If writes were performed in the order `A, B, C`, then a client sees either `A`, `A,B`, or `A,B,C`, but never out of order like `A,C` or `B,A,C`.
- Azure Cosmos DB accounts that are configured with consistent prefix consistency can associate any number of Azure regions with their Azure Cosmos DB account.

Eventual:

- Eventual consistency guarantees that in absence of any further writes, the replicas within the group 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.
- Azure Cosmos DB accounts that are configured with eventual consistency can associate any number of Azure regions with their Azure Cosmos DB account.
- The cost of a read operation (in terms of RUs consumed) with the eventual consistency level is the lowest of all the Azure Cosmos DB consistency levels.

Configuring the default consistency level

1. In the Azure portal, in the Jumpbar, click **Azure Cosmos DB**.
2. In the **Azure Cosmos DB** 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**.



Session consistency is most widely used consistency level both for single region as wel

It provides write latencies, availability and read throughput comparable to that of even
consistency guarantees that suit the needs of applications written to operate in the co

Click here, for more information on Azure Cosmos DB consistency levels.

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 an item to the Cosmos DB container. This enables the queries to honor the same consistency level as that of point reads. While Azure Cosmos DB is write optimized and supports sustained volumes of 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, consistent prefix, or eventual	Select from strong, bounded staleness, session, or eventual
Lazy	Select from strong, bounded staleness, session, consistent prefix, or eventual	Eventual
None	Select from strong, bounded staleness, session, consistent prefix, or eventual	Not applicable

As with read requests, you can lower the consistency level of a specific query request in each API.

Next steps

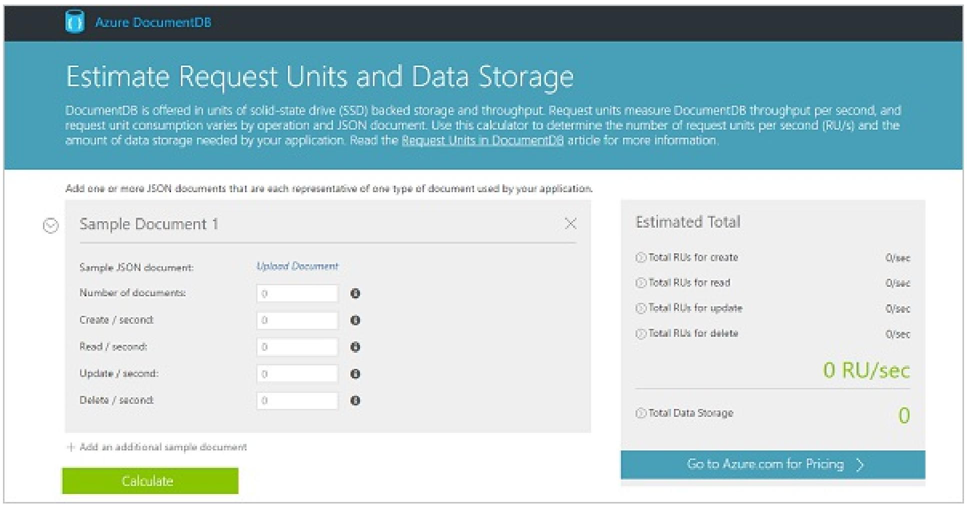
If you'd like to do more reading about consistency levels and tradeoffs, we recommend the following resources:

- Doug Terry. Replicated Data Consistency explained through baseball (video). https://www.youtube.com/watch?v=gluIh8zd26I
- 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
- Moni Naor , Avishai Wool, The Load, Capacity, and Availability of Quorum Systems, SIAM Journal on Computing, v.27 n.2, p.423-447, April 1998. http://epubs.siam.org/doi/abs/10.1137/S0097539795281232
- Sebastian Burckhardt, Chris Dern, Macanal Musuvathi, Roy Tan, Line-up: a complete and automatic linearizability checker, Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, June 05-10, 2010, Toronto, Ontario, Canada [doi>10.1145/1806596.1806634] http://dl.acm.org/citation.cfm?id=1806634
- Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein , Ion Stoica, Probabilistically bounded staleness for practical partial quorums, Proceedings of the VLDB Endowment, v.5 n.8, p.776-787, April 2012 http://dl.acm.org/citation.cfm?id=2212359

Request Units in Azure Cosmos DB

6/9/2017 • 14 min to read • Edit Online

Now available: Azure Cosmos DB request unit calculator. Learn more in Estimating your throughput needs.



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: 0

Create / second: 0

Read / second: 0

Update / second: 0

Delete / second: 0

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

+ Add an additional sample document
Calculate
Go to Azure.com for Pricing

Introduction

Azure Cosmos DB is Microsoft's globally distributed multi-model database. With Azure Cosmos DB, you don;t have to rent virtual machines, deploy software, or monitor databases. Azure Cosmos DB is operated and continuously monitored by Microsoft top engineers to deliver world class availability, performance, and data protection. You can access your data using APIs of your choice, as DocumentDB SQL (document), MongoDB (document), Azure Table Storage (key-value), and Gremlin (graph) are all natively supported. The currency of Azure Cosmos DB is the Request Unit (RU). With RU, you do not need to reserve read, write caprcities or provision CPU, Memory and IOPS.

Azure Cosmos DB supports a number of APIs with different operations ranging from reads, writes to complex graph queries. Since not all requests are equal, they are assigned a normalized amount of **request units** based on the amount of computation required to serve the request. The number of request units for an operation is deterministic, and you can track the number of request units consumed by any operation in Azure Cosmos DB via a response header.

To provide a predictable performance, you need to reserve throughput by unit of 100 RU/second. For each block of 100 RU/second, you can attach a block of 1,000 RU/minute. Combining provisioning per second and per minute is extremely powerful as you do not need to provision for peak and can save up to 75% in cost versus any service working only with per second provisioning.

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?

As Azure Cosmos DB is a multi-model database, this is important to note that we will refer to a collection/document for a document API, a graph/node for a graph API and a table/entity for table API. Throughput this document we will generalize to the concepts of container/item.

Request units and request charges

Azure Cosmos DB delivers fast, predictable performance by *reserving* resources to satisfy your application's throughput needs. Because application load and access patterns change over time, Azure Cosmos DB allows you to easily increase or decrease the amount of reserved throughput available to your application.

With Azure Cosmos DB, reserved throughput is specified in terms of request units processing per second or per minute (add-on). 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 or minute basis. Each operation in Azure Cosmos DB - 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. The query explorer is also a wonderful tool to test the core of a query.

We recommend getting started by watching the following video, where Aravind Ramachandran explains request units and predictable performance with Azure Cosmos DB.

Specifying request unit capacity in Azure Cosmos DB

When starting a new collection, table or graph, you specify the number of request units per second (RU per second) you want reserved. You can also decide if you want RU per minute enabled. If you enable it, you will get 10x what you get per second but per minute. Based on the provisioned throughput, Azure Cosmos DB allocates physical partitions to host your collection and splits/rebalances data across partitions as it grows.

Azure Cosmos DB requires a partition key to be specified when a collection is provisioned with 2,500 request units or higher. A partition key is also required to scale your collection's throughput beyond 2,500 request units in the future. Therefore, it is highly recommended to configure a partition key when creating a container regardless of your initial throughput. Since your data might have to be split across multiple partitions, it is necessary to pick a partition key that has a high cardinality (100 to millions of distinct values) so that your collection/table/graph and requests can be scaled uniformly by Azure Cosmos DB.

> NOTE
>
> A partition key is a logical boundary, and not a physical one. Therefore, you do not need to limit the number of distinct partition key values. It is in fact better to have more distinct partition key values than less, as Azure Cosmos DB has more load balancing options.

Here is a code snippet for creating a collection with 3,000 request units per second using the .NET SDK:

```
DocumentCollection myCollection = new DocumentCollection();
myCollection.Id = "coll";
myCollection.PartitionKey.Paths.Add("/deviceId");

await client.CreateDocumentCollectionAsync(
   UriFactory.CreateDatabaseUri("db"),
   myCollection,
   new RequestOptions { OfferThroughput = 3000 });
```

Azure Cosmos DB operates on a reservation model on throughput. That is, you are billed for the amount of throughput *reserved*, regardless of how much of that throughput is actively *used*. As your application's load, data, and usage patterns change you can easily scale up and down the amount of reserved RUs through SDKs or using the Azure Portal.

Each collection/table/graph are mapped to an `Offer` resource in Azure Cosmos DB, which has metadata about the provisioned throughput. You can change the allocated throughput by looking up the corresponding offer resource for a container, then updating it with the new throughput value. Here is a code snippet for changing the throughput of a collection to 5,000 request units per second using the .NET SDK:

```
// 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);
```

There is no impact to the availability of your container when you change the throughput. Typically the new reserved throughput is effective within seconds on application of the new throughput.

Request unit considerations

When estimating the number of request units to reserve for your Azure Cosmos DB container, it is important to take the following variables into consideration:

- **Item size**. As size increases the units consumed to read or write the data will also increase.
- **Item property count**. Assuming default indexing of all properties, the units consumed to write a document/node/ntity 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 items.
- **Indexed properties**. An index policy on each container 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 item is automatically indexed, you will consume fewer request units if you choose not to index some of your items.
- **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 item consisting of 10 unique property values (excluding system properties). A request to create (insert), replace or delete the same item will consume more processing from the service and thereby more request units.

> NOTE
>
> The baseline of 1 request unit for a 1KB item corresponds to a simple GET by self link or id of the item.

For example, here's a table that shows how many request units to provision at three different item 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.

Item size	**Reads/second**	**Writes/second**	**Request units**
1 KB	500	100	(500 * 1) + (100 * 5) = 1,000 RU/s
1 KB	500	500	(500 * 1) + (500 * 5) = 3,000 RU/s
4 KB	500	100	(500 * 1.3) + (100 * 7) = 1,350 RU/s
4 KB	500	500	(500 * 1.3) + (500 * 7) = 4,150 RU/s
64 KB	500	100	(500 * 10) + (100 * 48) = 9,800 RU/s
64 KB	500	500	(500 * 10) + (500 * 48) = 29,000 RU/s

Use the request unit calculator

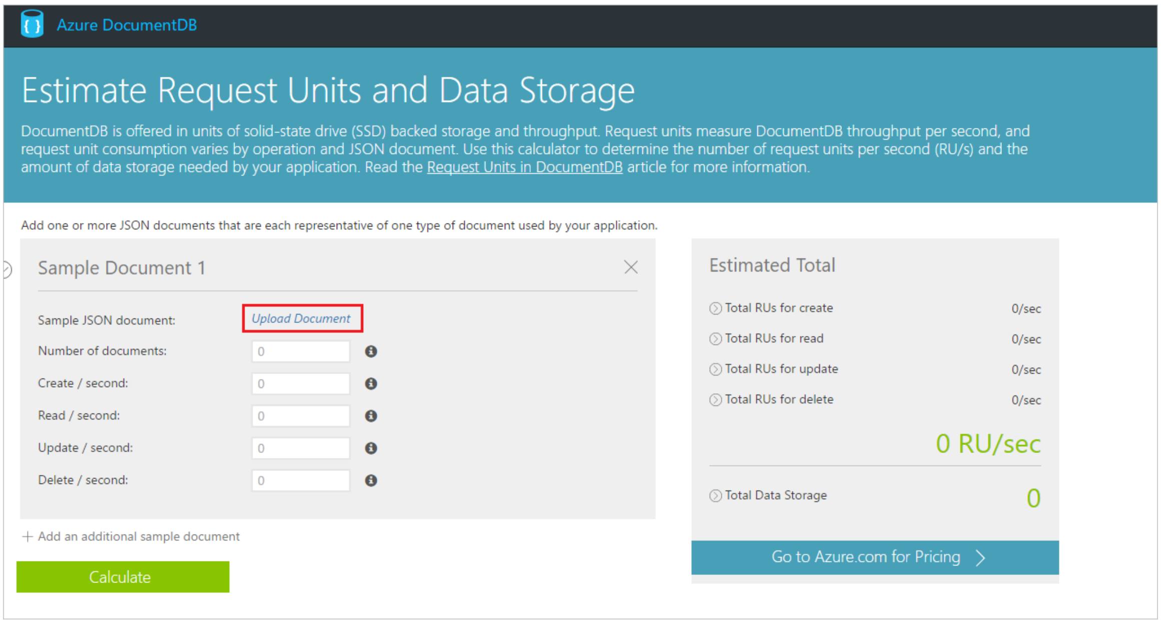
To help customers fine tune their throughput estimations, there is a web based request unit calculator to help estimate the request unit requirements for typical operations, including:

- Item creates (writes)
- Item reads
- Item deletes
- Item updates

The tool also includes support for estimating data storage needs based on the sample items you provide.

Using the tool is simple:

1. Upload one or more representative items.



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:	0
Create / second:	0
Read / second:	0
Update / second:	0
Delete / second:	0

+ Add an additional sample document

Calculate

Estimated Total

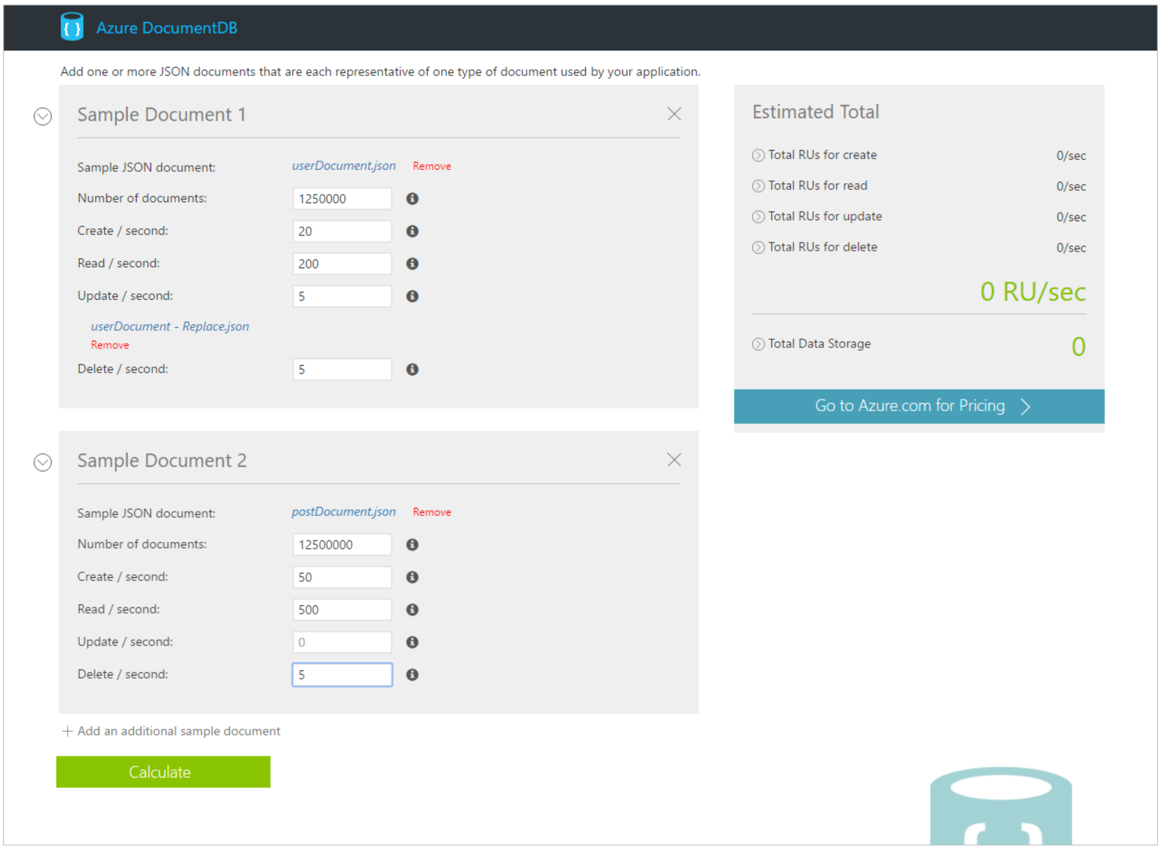
Total RUs for create	0/sec
Total RUs for read	0/sec
Total RUs for update	0/sec
Total RUs for delete	0/sec

0 RU/sec

Total Data Storage 0

Go to Azure.com for Pricing

2. To estimate data storage requirements, enter the total number of items you expect to store.
3. Enter the number of items create, read, update, and delete operations you require (on a per-second basis). To estimate the request unit charges of item update operations, upload a copy of the sample item from step 1 above that includes typical field updates. For example, if item updates typically modify two properties named lastLogin and userVisits, then simply copy the sample item, update the values for those two properties, and upload the copied item.



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

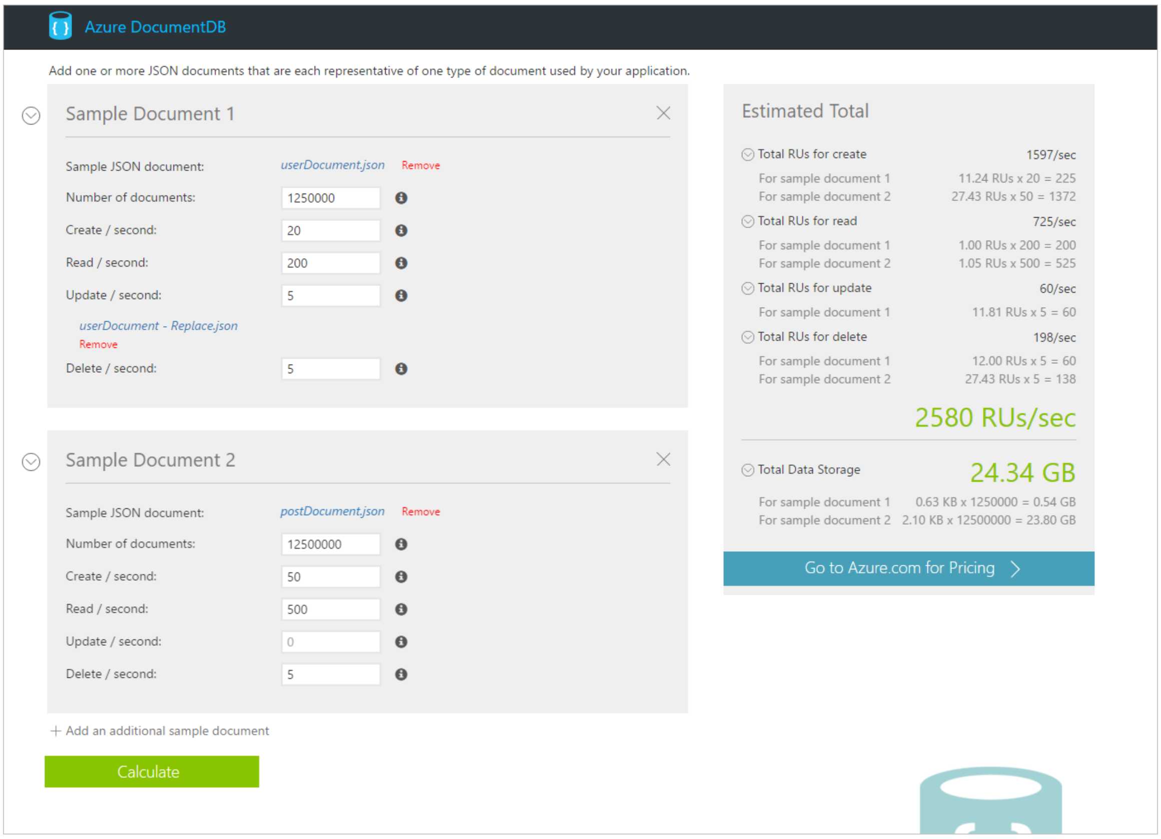
Total RUs for create	0/sec
Total RUs for read	0/sec
Total RUs for update	0/sec
Total RUs for delete	0/sec

0 RU/sec

Total Data Storage 0

Go to Azure.com for Pricing

4. Click calculate and examine the results.



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

Estimated Total

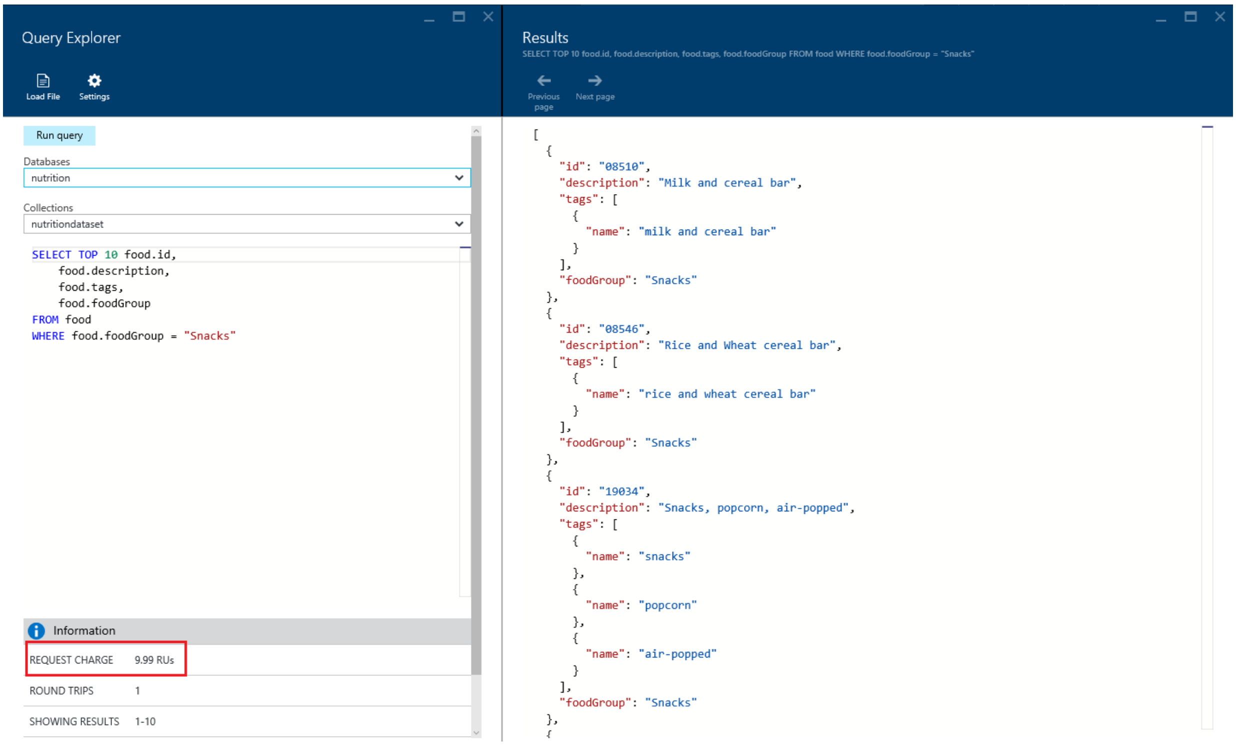
Total RUs for create	1597/sec
For sample document 1	11.24 RUs x 20 = 225
For sample document 2	27.43 RUs x 50 = 1372
Total RUs for read	725/sec
For sample document 1	1.00 RUs x 200 = 200
For sample document 2	1.05 RUs x 500 = 525
Total RUs for update	60/sec
For sample document 1	11.81 RUs x 5 = 60
Total 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

NOTE

If you have item types which will differ dramatically in terms of size and the number of indexed properties, then upload a sample of each *type* of typical item to the tool and then calculate the results.

Use the Azure Cosmos DB request charge response header

Every response from the Azure Cosmos DB 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 Azure Cosmos DB Query Explorer in the Azure portal provides request charge information for executed queries.



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

SELECT TOP 10 food.id, food.description, food.tags, food.foodGroup FROM food WHERE food.foodGroup = "Snacks"

```
[
  {
    "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 item 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 Azure Cosmos DB script usage as well.

> NOTE
>
> If you have item 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 item.

For example:

1. Record the request unit charge of creating (inserting) a typical item.
2. Record the request unit charge of reading a typical item.
3. Record the request unit charge of updating a typical item.
4. Record the request unit charge of typical, common item 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.

Use API for MongoDB's GetLastRequestStatistics command

API for MongoDB supports a custom command, *getLastRequestStatistics*, for retrieving the request charge for specified operations.

For example, in the Mongo Shell, execute the operation you want to verify the request charge for.

```
> db.sample.find()
```

Next, execute the command *getLastRequestStatistics*.

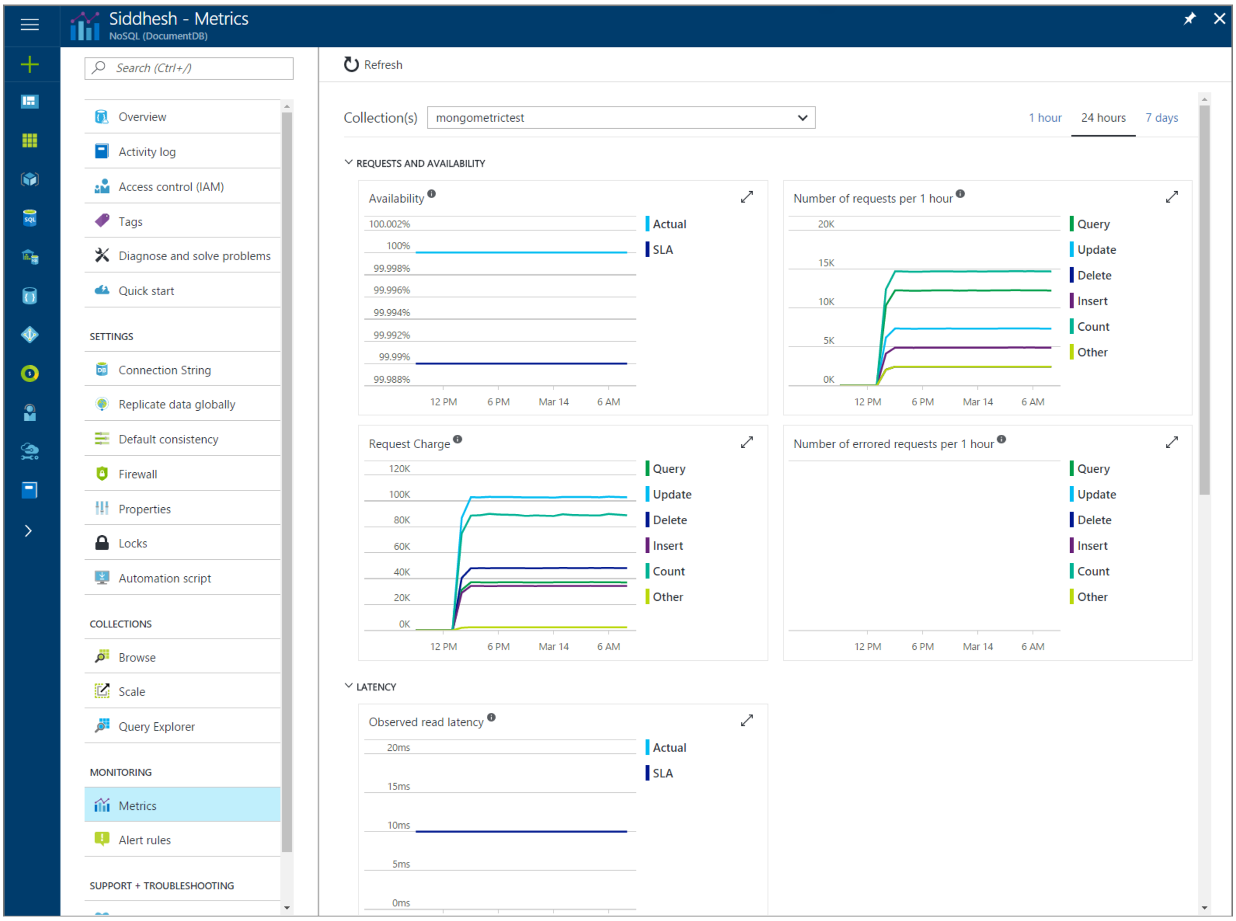
```
> db.runCommand({getLastRequestStatistics: 1})
{
    "_t": "GetRequestStatisticsResponse",
    "ok": 1,
    "CommandName": "OP_QUERY",
    "RequestCharge": 2.48,
    "RequestDurationInMilliSeconds" : 4.0048
}
```

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 item used by your application and then estimating the number of operations you anticipate performing each second.

> NOTE
>
> If you have item 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 item.

Use API for MongoDB's portal metrics

The simplest way to get a good estimation of request unit charges for your API for MongoDB database is to use the Azure portal metrics. With the *Number of requests* and *Request Charge* charts, you can get an estimation of how many request units each operation is consuming and how many request units they consume relative to one another.



A request unit estimation example

Consider the following ~1KB document:

```json
{
 "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
>
> Documentss are minified in Azure Cosmos DB, 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 item (the approximate request unit charge assumes that the account consistency level is set to “Session” and that all items are automatically indexed):

OPERATION	REQUEST UNIT CHARGE
Create item	~15 RU

OPERATION	REQUEST UNIT CHARGE
Read item	~1 RU
Query item 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 ITEMS
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 items 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 item	10	150
Read item	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 in Azure Cosmos DB

Recall that request unit consumption is evaluated as a rate per second if Request Unit per Minute is disabled or the budget is empty. For applications that exceed the provisioned request unit rate for a container, 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 container.

Exceeding reserved throughput limits in API for MongoDB

Applications that exceed the provisioned request units for a collection will be throttled until the rate drops below the reserved level. When a throttle occurs, the backend will preemptively end the request with a *16500* error code - *Too Many Requests*. By default, API for MongoDB will automatically retry up to 10 times before returning a *Too Many Requests* error code. If you are receiving many *Too Many Requests* error codes, you may consider either adding retry behavior 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 Cosmos DB databases, explore these resources:

- Azure Cosmos DB pricing
- Partitioning data in Azure Cosmos DB

To learn more about Azure Cosmos DB, see the Azure Cosmos DB documentation.

To get started with scale and performance testing with Azure Cosmos DB, see Performance and Scale Testing with Azure Cosmos DB.

Request units per minute in Azure Cosmos DB

6/13/2017 • 8 min to read • Edit Online

Azure Cosmos DB is designed to help you achieve a fast, predictable performance and scale seamlessly along with your application's growth. You can provision throughput on a Cosmos DB container at both, per-second and at per-minute (RU/m) granularities. The provisioned throughput at per-minute granularity is used to manage unexpected spikes in the workload occurring at a per-second granularity.

This article provides an overview of how the provisioning of Request Unit per Minute (RU/m) works. The goal in mind with provisioning of RU/m is to provide a predictable performance around unpredictable needs (especially if you need to run analytics on top of your operational data) and spiky workloads. We want to have our customers consume more the throughput they provision so they can scale quickly with peace of mind.

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

- How does a Request Unit per Minute work?
- What is the difference between Request Unit per Minute and Request Unit per Second?
- How to provision RU/m?
- Under which scenario shall I consider provisioning Request Unit per Minute?
- How to use the portal metrics to optimize my cost and performance?
- Define which type of request can consume your RU/m budget?

Provisioning request units per minute (RU/m)

When you provision Azure Cosmos DB at the second granularity (RU/s), you get the guarantee that your request succeeds at a low latency if your throughput has not exceeded the capacity provisioned within that second. With RU/m, the granularity is at the minute with the guarantee that your request succeeds within that minute. Compared to bursting systems, we make sure that the performance you get is predictable and you can plan on it.

The way per minute provisioning works is simple:

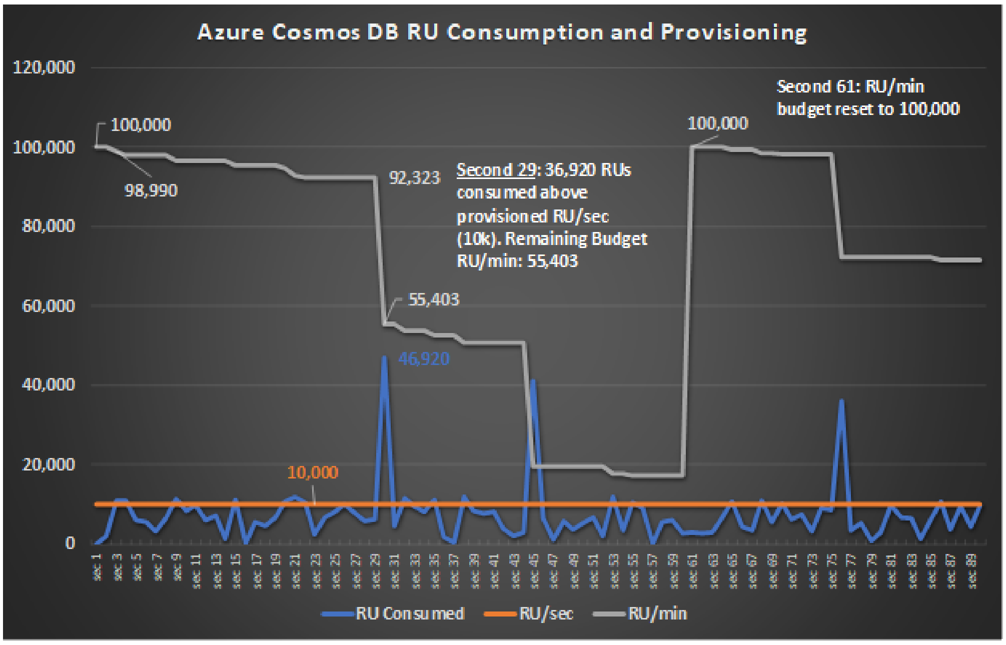
- RU/m is billed hourly and in addition to RU/s. For more details, please visit Azure Cosmos DB pricing page.
- RU/m can be enabled at collection level. That can be done through the SDKs (Node.js, Java, or .Net) or through the portal (also include MongoDB API workloads)
- When RU/m is enabled, for every 100 RU/s provisioned, you also get 1,000 RU/m provisioned (the ratio is 10x)
- At a given second, a request unit consumes your RU/m provisioning only if you have exceeded your per second provisioning within that second
- Once the 60-second period (UTC) ends, the per minute provisioning is refilled
- RU/m can be enabled only for collections with a maximum provisioning of 5,000 RU/s per partition. If you scale your throughput needs and have such a high level of provisioning per partition, you will get a warning message

Below is a concrete example, in which a customer can provision 10kRU/s with 100kRU/m, saving 73% in cost against provisioning for peak (at 50kRU/sec) through a 90-second period on a collection that has 10,000 RU/s and 100,000 RU/m provisioned:

- 1st second: The RU/m budget is set at 100,000
- 3rd second: During that second the consumption of Request Unit was 11,010 RUs, 1,010 RUs above the RU/s provisioning. Therefore, 1,010 RUs are deducted from the RU/m budget. 98,990 RUs are available for the next 57 seconds in the RU/m budget
- 29th second: During that second, a large spike happened (>4x higher than provisioning per second) and the

consumption of Request Unit was 46,920 RUs. 36,920 RUs are deducted from the RU/m budget that dropped from 92,323 RUs (28th second) to 55,403 RUs (29th second)

- 61st second: RU/m budget is set back to 100,000 RUs.

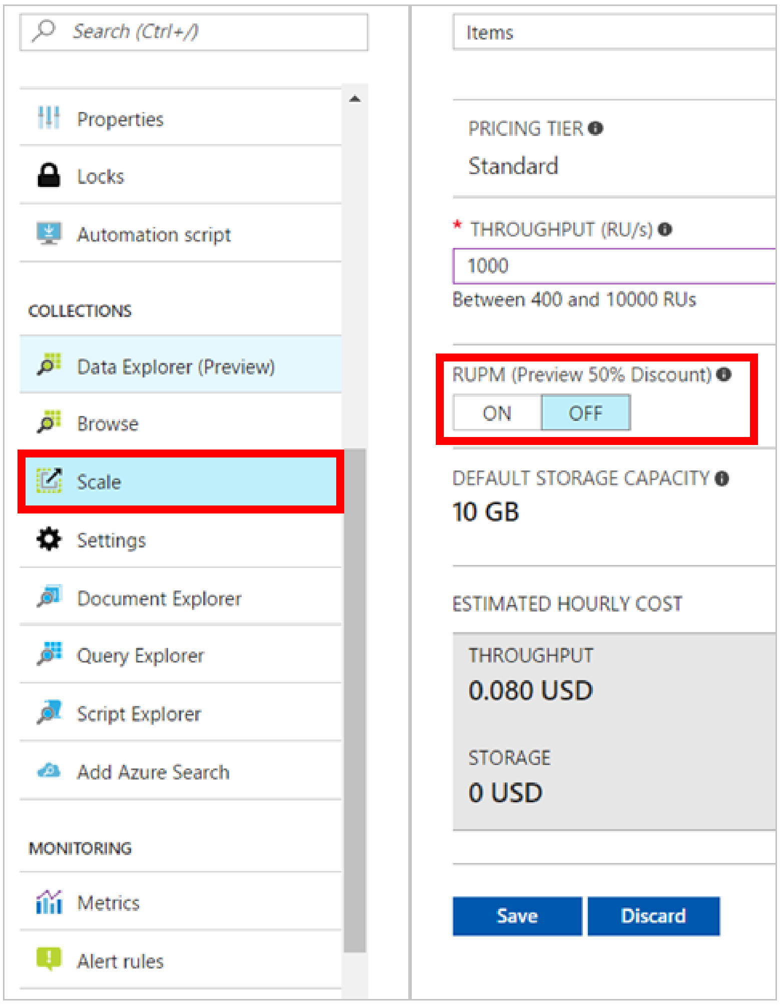


Specifying request unit capacity with RU/m

When creating an Azure Cosmos DB collection, you specify the number of request units per second (RU per second) you want reserved for the collection. You can also decide if you want to add RU per minute. This can be done through the Portal or the SDK.

Through the Portal

Enabling or disabling RU per minute simply requires a click when provisioning a collection.



Items

PRICING TIER
Standard

* THROUGHPUT (RU/s)
1000
Between 400 and 10000 RUs

RUPM (Preview 50% Discount)
ON [✓ OFF]

DEFAULT STORAGE CAPACITY
10 GB

ESTIMATED HOURLY COST

THROUGHPUT
0.080 USD

STORAGE
0 USD

Through the SDK

First, this is important to note that RU/m is only available for the following SDKs:

- .Net 1.14.0
- Java 1.11.0
- Node.js 1.12.0
- Python 2.2.0

Here is a code snippet for creating a collection with 3,000 request units per second and 30,000 request units per minute using the .NET SDK:

```
// Create a collection with RU/m enabled
DocumentCollection myCollection = new DocumentCollection();
myCollection.Id = "coll";
myCollection.PartitionKey.Paths.Add("/deviceId");

// Set the throughput to 3,000 request units per second which will give you 30,000 request units per minute as the RU/m budget
await client.CreateDocumentCollectionAsync(
   UriFactory.CreateDatabaseUri("db"),
   myCollection,
   new RequestOptions { OfferThroughput = 3000, OfferEnableRUPerMinuteThroughput = true });
```

Here is a code snippet for changing the throughput of a collection to 5,000 request units per second without provisioning RU per minute using the .NET SDK:

```
// Get the current offer
Offer offer = client.CreateOfferQuery()
    .Where(r => r.ResourceLink == collection.SelfLink)
    .AsEnumerable()
    .SingleOrDefault();

// Set the throughput to 5000 request units per second without RU/m enabled (the last parameter to OfferV2 constructor below)
OfferV2 offerV2 = new OfferV2(offer, 5000, false);

// Now persist these changes to the database by replacing the original resource
await client.ReplaceOfferAsync(offerV2);
```

Good fit scenarios

In this section, we provide an overview of scenarios that are a good fit for enabling request units per minute.

Dev/Test environment: Good fit. During the development stage, if you are testing your application with different workloads, RU/m can provide the flexibility at this stage. While the emulator is a great free tool to test Azure Cosmos DB. However if you want to start in a cloud environment, you will have a great flexibility with RU/m for your adhoc performance needs. You will spend more time developing, less worrying about performance needs at first. We recommend starting with the minimum RU/s provisioning and enable RU/m.

Unpredictable, spiky, minute granularity needs: Good fit – Savings: 25-75%. We have seen large improvement from RU/m and most production scenarios are into that group. If you have an IoT workload that has spike a few times in a minute, if you have queries running when your system makes mass insert at the same time, you will need extra capacity for handeling spiky needs. We recommend optimizing your resource needs by applying our step by step approach below.

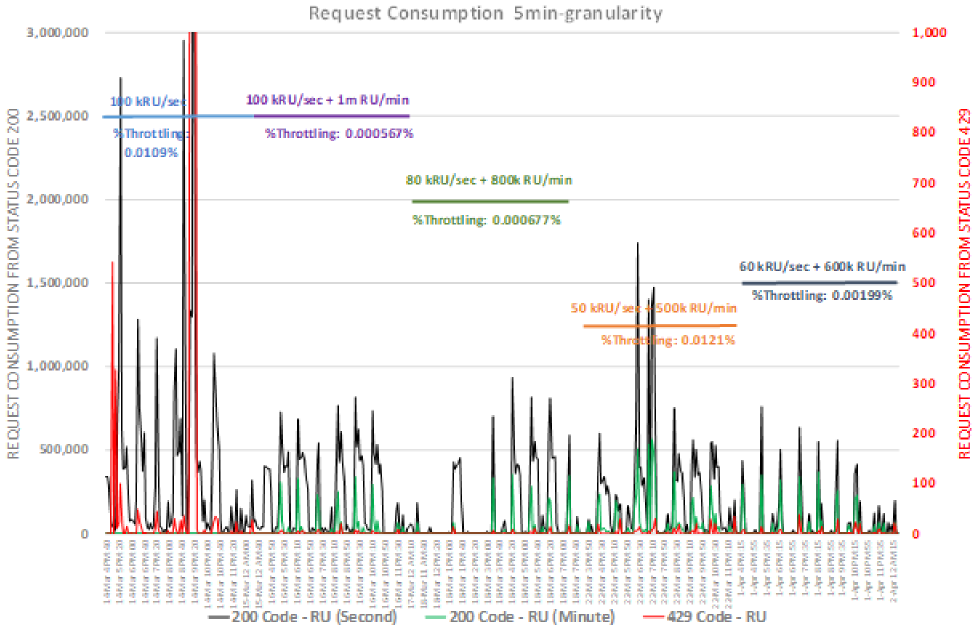


Figure - RU consumption benchmark

Peace of mind: Good fit – Savings: 10-20%. Sometimes, you just want to have peace of mind and not worry about potential peaks and throttling. This feature is the right one for you. In that case, we recommend enabling RU/m and slightly lower your per second provisioning. This case is different from the above as you will not try to optimize aggressively your provisioning. This is more of a "Zero Throttling" mindset you are in.

Critical operations with adhoc needs: We sometimes recommend to only let critical operations access RU/m budget so the budget doesn't get consume by adhoc or less important operations. That can be easily defined in the section below.

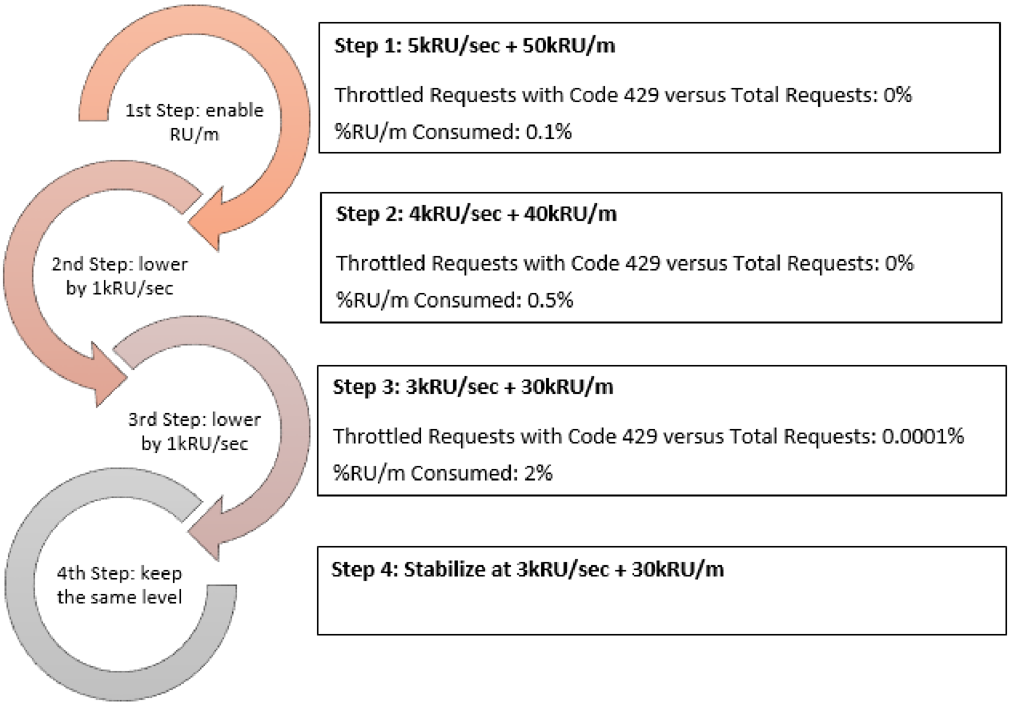
Using the portal metrics to optimize cost and performance

In the coming weeks, we will further develop the content around monitoring RUs minute consumption to optimize your throughput needs.

Through the portal metrics, you can see how much of regular RU seconds you consume versus RU minutes. Monitoring these metrics should help you optimize your provisioning.

We recommend a step by step approach on how to use RU/m to your advantage. For each step, you should have an overview of the RU consumption representing a full cycle of your workload (it could be hours, days, or even weeks) and get insights on the utilization of what you provision.

The principle behind this approach is to make your throughput provisioning as close as possible to a provisioning point that matches your performance criteria below.


1st Step: enable RU/m
2nd Step: lower by 1kRU/sec
3rd Step: lower by 1kRU/sec
4th Step: keep the same level

Step 1: 5kRU/sec + 50kRU/m

Throttled Requests with Code 429 versus Total Requests: 0%

%RU/m Consumed: 0.1%

Step 2: 4kRU/sec + 40kRU/m

Throttled Requests with Code 429 versus Total Requests: 0%

%RU/m Consumed: 0.5%

Step 3: 3kRU/sec + 30kRU/m

Throttled Requests with Code 429 versus Total Requests: 0.0001%

%RU/m Consumed: 2%

Step 4: Stabilize at 3kRU/sec + 30kRU/m

To understand the optimal provisioning point for your workload, you need to understand:

- Consumption patterns: no, infrequent or sustained spikes? Small (2x average), medium, or large (>10x average) spikes?
- Percent of throttled requests: do you feel comfortable if you have a bit of throttling? If so, by how much?

Once you have identified what your goals are, you will be able to get closer to the optimal provisioning.

To assist you, we want to provide an overall guidance on how to optimize your provisioning based on your RU/m consumption. This guidance doesn’t apply to all kind of workloads but is based on the private preview knowledge. We might change such baselines as we learn more:

RU/M % UTILIZATION	DEGREE OF UTILIZATION OF RU/M	RECOMMENDED ACTIONS FOR PROVISIONING
0-1%	Under utilization	Lower RU/s to consume more RU/m
1-10%	Healthy use	Keep the same provisioning level
Above 10%	Over utilization	Increase RU/s to rely less on RU/m

Select which operations can consume the RU/m budget

At request level, you can also enable/disable RU/m budget to serve the request irrespective of operation type. If regular provisioned RUs/sec budget is consumed and the request cannot consume the RU/m budget, this request will be throttled. By default, any request is served by RU/m budget if RU/m throughput budget is activated.

Here is a code snippet for disabling RU/m budget using the DocumentDB API for CRUD and query operations.

```
// In order to disable any CRUD request for RU/m, set DisableRUPerMinuteUsage to true in RequestOptions
await client.CreateDocumentAsync(
   UriFactory.CreateDocumentCollectionUri("db", "container"),
   new Document { Id = "Cosmos DB" },
   new RequestOptions { DisableRUPerMinuteUsage = true });
// In order to disable any query request for RU/m, set DisableRUPerMinuteOnRequest to true in RequestOptions
FeedOptions feedOptions = new FeedOptions();
feedOptions.DisableRUPerMinuteUsage = true;
var query = client.CreateDocumentQuery<Book>(
   UriFactory.CreateDocumentCollectionUri("db", "container"),
   "select * from c",feedOptions).AsDocumentQuery();
```

Next steps

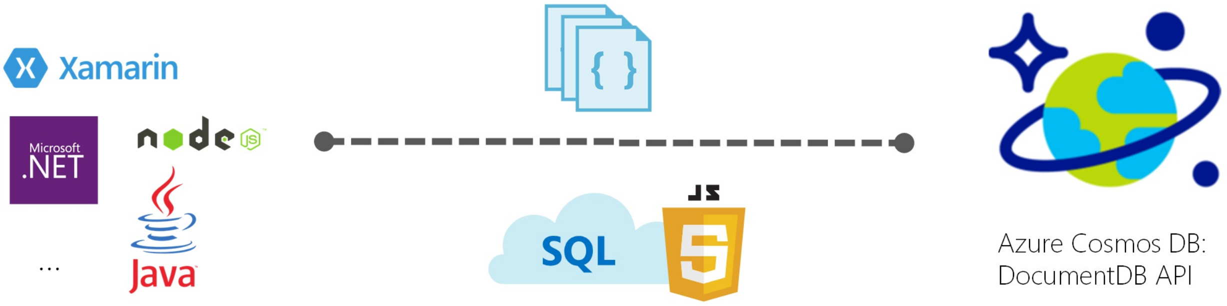
In this article, we've described how partitioning works in Azure Cosmos DB, how you can create partitioned collections, and how you can pick a good partition key for your application.

- Perform scale and performance testing with Azure Cosmos DB. See Performance and Scale Testing with Azure Cosmos DB for a sample.
- Get started coding with the SDKs or the REST API.
- Learn about provisioned throughput in Azure Cosmos DB

Introduction to Azure Cosmos DB: DocumentDB API

6/9/2017 • 7 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed, multi-model database service for mission-critical applications. Azure Cosmos DB provides turn-key global distribution, elastic scaling of throughput and storage worldwide, single-digit millisecond latencies at the 99th percentile, five well-defined consistency levels, and guaranteed high availability, all backed by industry-leading SLAs. Azure Cosmos DB automatically indexes data without requiring you to deal with schema and index management. It is multi-model and supports document, key-value, graph, and columnar data models.



With the DocumentDB API, Azure Cosmos DB provides rich and familiar SQL query capabilities with consistent low latencies over schema-less JSON data. In this article, we provide an overview of the Azure Cosmos DB's DocumentDB API, and how you can use it to store massive volumes of JSON data, query them within order of milliseconds latency, and evolve the schema easily.

What capabilities and key features does Azure Cosmos DB offer?

Azure Cosmos DB, via the DocumentDB API, offers the following key capabilities and benefits:

- **Elastically scalable throughput and storage:** Easily scale up or scale down your JSON database to meet your application needs. Your data is stored on solid state disks (SSD) for low predictable latencies. Azure Cosmos DB supports containers for storing JSON data called collections that can scale to virtually unlimited storage sizes and provisioned throughput. You can elastically scale Azure Cosmos DB with predictable performance seamlessly as your application grows.

- **Multi-region replication:** Azure Cosmos DB transparently replicates your data to all regions you've associated with your Azure Cosmos DB account, enabling you to develop applications that require global access to data while providing tradeoffs between consistency, availability and performance, all with corresponding guarantees. Azure Cosmos DB 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 Azure Cosmos DB.

- **Ad hoc queries with familiar SQL syntax:** Store heterogeneous JSON documents and query these documents through a familiar SQL syntax. Azure Cosmos DB 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 Azure Cosmos DB.

- **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. The DocumentDB API 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 five well defined consistency levels to achieve optimal trade-off between consistency and performance. For queries and read operations, Azure Cosmos DB offers five distinct consistency levels: strong, bounded-staleness, session, consistent prefix, 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.
- **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 an Azure Cosmos DB 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 the DocumentDB API 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. The DocumentDB API embraces existing formats, languages, and standards while offering high value database capabilities on top of them.
- **Automatic indexing:** By default, Azure Cosmos DB 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.

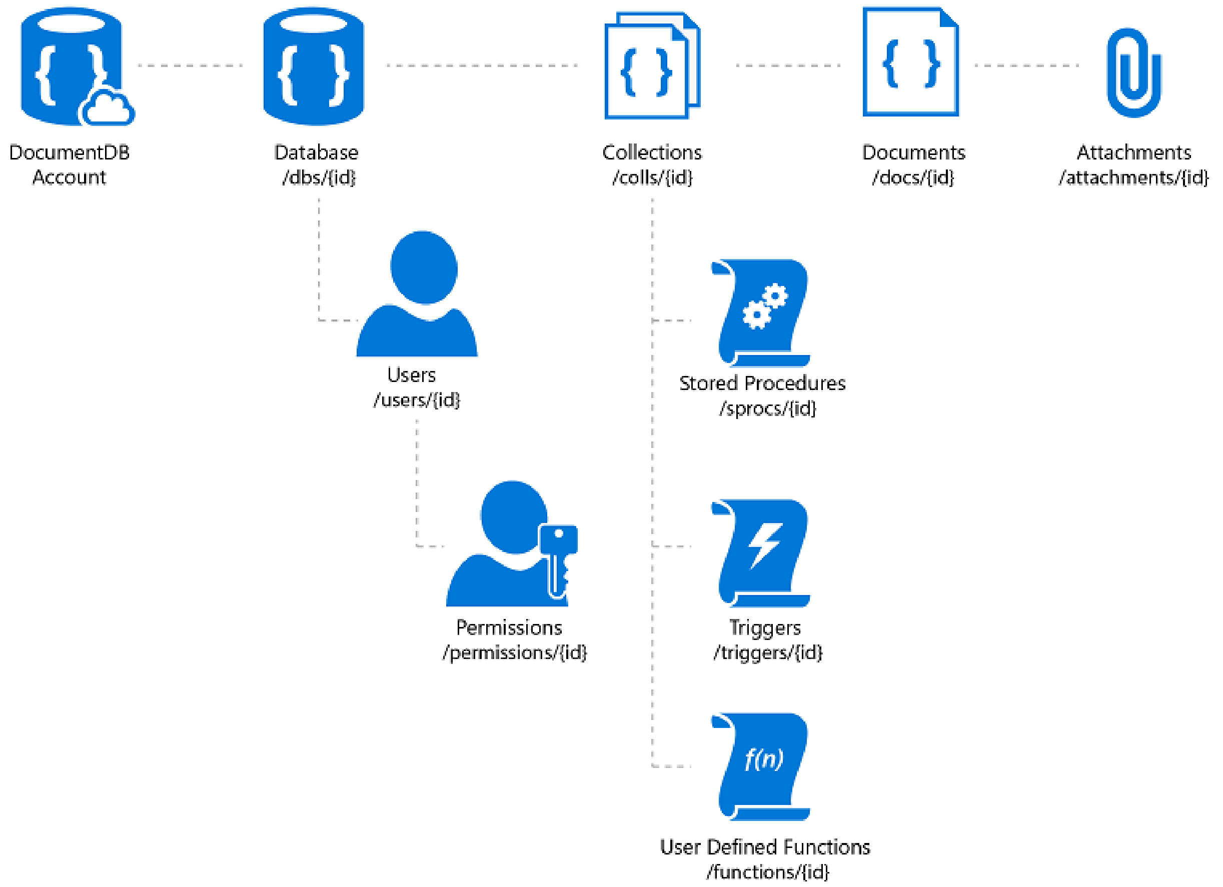
How do you manage data with the DocumentDB API?

The DocumentDB API helps 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 Azure Cosmos DB database account is a unique namespace that gives you access to Azure Cosmos DB. 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 Azure Cosmos DB 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 Azure Cosmos DB 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.

NOTE

Since the DocumentDB API was previously available as the Azure DocumentDB service, you can continue to provision, monitor, and manage accounts created via the Azure Resource Management REST API or tools using either the Azure DocumentDB or Azure Cosmos DB resource names. We use the names interchangeably when referring to the Azure DocumentDB APIs.

How can I develop apps with the DocumentDB API?

Azure Cosmos DB exposes resources through the DocumentDB REST API that can be called by any language capable of making HTTP/HTTPS requests. Additionally, we offer programming libraries for several popular languages for the DocumentDB API. The client libraries simplify many aspects of working with the API 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

DOWNLOAD	DOCUMENTATION
Java SDK	Java library
JavaScript SDK	JavaScript library
n/a	Server-side JavaScript SDK
Python SDK	Python library
n/a	API for MongoDB

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

Beyond basic create, read, update, and delete operations, the DocumentDB API 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

The DocumentDB API 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 the DocumentDB API 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, the DocumentDB .NET SDK also offers a LINQ query provider.

Transactions and JavaScript execution

The DocumentDB API 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 the DocumentDB API 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.

Are there any online courses on Azure Cosmos DB?

Yes, there's a Microsoft Virtual Academy course on Azure DocumentDB.

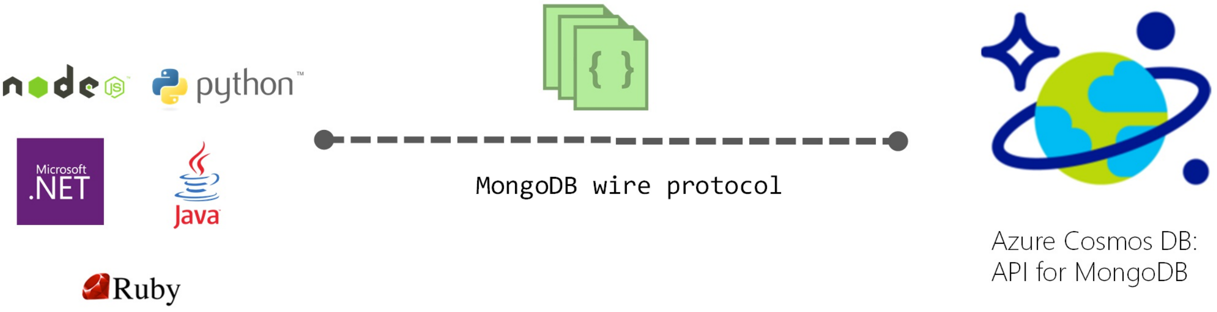
Next steps

Already have an Azure account? Then you can get started with Azure Cosmos DB by following our quick starts, which will walk you through creating an account and getting started with Cosmos DB.

Introduction to Azure Cosmos DB: API for MongoDB

5/30/2017 • 3 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed, multi-model database service for mission-critical applications. Azure Cosmos DB provides turn-key global distribution, elastic scaling of throughput and storage worldwide, single-digit millisecond latencies at the 99th percentile, five well-defined consistency levels, and guaranteed high availability, all backed by industry-leading SLAs. Azure Cosmos DB automatically indexes data without requiring you to deal with schema and index management. It is multi-model and supports document, key-value, graph, and columnar data models.



Cosmos DB databases can be used as the data store for apps written for MongoDB. This means that by using existing drivers, your application written for MongoDB can now communicate with Cosmos DB and use Cosmos DB databases instead of MongoDB databases. In many cases, you can switch from using MongoDB to Cosmos DB by simply changing a connection string. Using this functionality, you can easily build and run MongoDB database applications in the Azure cloud with Azure Cosmos DB's global distribution and comprehensive industry leading SLAs, while continuing to use familiar skills and tools for MongoDB.

What is the benefit of using Azure Cosmos DB for MongoDB applications?

Elastically scalable throughput and storage: Easily scale up or down your MongoDB database to meet your application needs. Your data is stored on solid state disks (SSD) for low predictable latencies. Cosmos DB supports MongoDB collections that can scale to virtually unlimited storage sizes and provisioned throughput. You can elastically scale Cosmos DB with predictable performance seamlessly as your application grows.

Multi-region replication: Cosmos DB transparently replicates your data to all regions you've associated with your MongoDB account, enabling you to develop applications that require global access to data while providing tradeoffs between consistency, availability and performance, all with corresponding guarantees. Cosmos DB 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.

MongoDB compatibility: You can use your existing MongoDB expertise, application code, and tooling. You can develop applications using MongoDB and deploy them to production using the fully managed globally distributed Cosmos DB service.

No server management: You don't have to manage and scale your MongoDB databases. Cosmos DB is a fully managed service, which means you do not have to manage any infrastructure or Virtual Machines yourself. Cosmos DB is available in 30+ Azure Regions.

Tunable consistency levels: Select from five well defined consistency levels to achieve optimal trade-off between consistency and performance. For queries and read operations, Cosmos DB offers five distinct consistency levels: strong, bounded-staleness, session, consistent prefix, 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.

Automatic indexing: By default, Cosmos DB automatically indexes all the properties within documents in your MongoDB database and does not expect or require any schema or creation of secondary indices.

Enterprise grade - Azure Cosmos DB supports multiple local replicas to deliver 99.99% availability and data protection in the face of local and regional failures. Azure Cosmos DB has enterprise grade compliance certifications and security features.

Learn more in this Azure Friday video with Scott Hanselman and Azure Cosmos DB Principal Engineering Manager, Kirill Gavrylyuk.

How to get started

Follow the MongoDB quickstarts to create a Cosmos DB account and migrate your existing Mongo DB application to use Cosmos DB, or build a new one:

- Migrate an existing Node.js MongoDB web app.
- Build a MongoDB API web app with .NET and the Azure portal
- Build a MongoDB API console app with Java and the Azure portal

Next steps

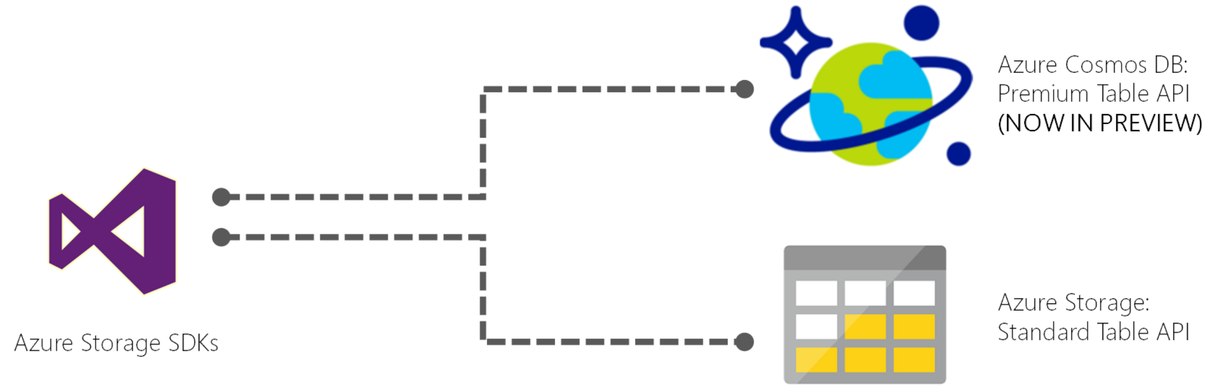
Information about Azure Cosmos DB's MongoDB API is integrated into the overall Azure Cosmos DB documentation, but here are a few pointers to get you started:

- Follow the Connect to a MongoDB account tutorial to learn how to get your account connection string information.
- Follow the Use MongoChef with Azure Cosmos DB tutorial to learn how to create a connection between your Azure Cosmos DB database and MongoDB app in MongoChef.
- Follow the Migrate data to Azure Cosmos DB with protocol support for MongoDB tutorial to import your data to an API for MongoDB database.
- Connect to an API for MongoDB account using Robomongo.
- Learn how many RUs your operations are using with the GetLastRequestStatistics command and the Azure portal metrics.
- Learn how to configure read preferences for globally distributed apps.

Introduction to Azure Cosmos DB: Table API

6/9/2017 • 2 min to read • Edit Online

Azure Cosmos DB is Microsoft's globally distributed, multi-model database service for mission-critical applications. Azure Cosmos DB provides turn-key global distribution, elastic scaling of throughput and storage worldwide, single-digit millisecond latencies at the 99th percentile, five well-defined consistency levels, and guaranteed high availability, all backed by industry-leading SLAs. Azure Cosmos DB automatically indexes data without requiring you to deal with schema and index management. It is multi-model and supports document, key-value, graph, and columnar data models.



Azure Cosmos DB provides the Table API (preview) for applications that need a key-value store with flexible schema, predictable performance, global distribution, and high throughput. The Table API provides the same functionality as Azure Table storage, but leverages the benefits of the Azure Cosmos DB engine.

You can continue to use Azure Table storage for tables with high storage and lower throughput requirements. Azure Cosmos DB will introduce support for storage-optimized tables in a future update, and existing and new Azure Table storage accounts will be upgraded to Azure Cosmos DB.

Premium and standard Table APIs

If you currently use Azure Table storage, you gain the following benefits by moving to Azure Cosmos DB's "premium table" preview:

	AZURE TABLE STORAGE	AZURE COSMOS DB: TABLE STORAGE (PREVIEW)
Latency	Fast, but no upper bounds on latency	Single-digit millisecond latency for reads and writes, backed with <10 ms latency reads and <15 ms latency writes at the 99th percentile, at any scale, anywhere in the world
Throughput	Highly scalable, but no dedicated throughput model. Tables have a scalability limit of 20,000 operations/s	Highly scalable with dedicated reserved throughput per table, that is backed by SLAs. Accounts have no upper limit on throughput, and support >10 million operations/s per table

	AZURE TABLE STORAGE	AZURE COSMOS DB: TABLE STORAGE (PREVIEW)
Global Distribution	Single region with one optional readable secondary read region for HA. You cannot initiate failover	Turn-key global distribution from one to 30+ regions, Support for automatic and manual failovers at any time, anywhere in the world
Indexing	Only primary index on PartitionKey and RowKey. No secondary indexes	Automatic and complete indexing on all properties, no index management
Query	Query execution uses index for primary key, and scans otherwise.	Queries can take advantage of automatic indexing on properties for fast query times. Azure Cosmos DB's database engine is capable of supporting aggregates, geo-spatial, and sorting.
Consistency	Strong within primary region, Eventual with secondary region	five well-defined consistency levels to trade off availability, latency, throughput, and consistency based on your application needs
Pricing	Storage-optimized	Throughput-optimized
SLAs	99.9% availability	99.99% availability within a single region, and ability to add more regions for higher availability. Industry-leading comprehensive SLAs on general availability

How to get started

Create an Azure Cosmos DB account in the Azure portal, and get started with our Quickstart for Table API using .NET.

Next steps

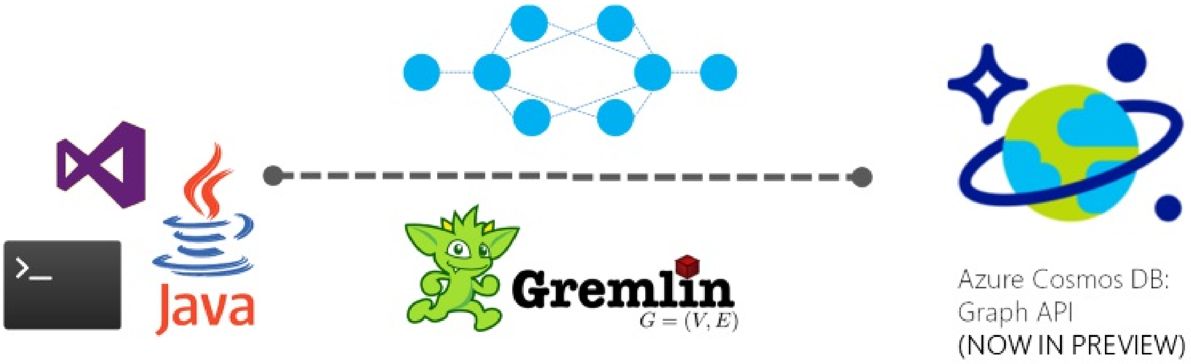
Here are a few pointers to get you started:

- Get started with Azure Cosmos DB's Table API using existing NET Table SDK.
- Learn about Global distribution with Azure Cosmos DB.
- Learn about Provisioned throughput in Azure Cosmos DB.

Introduction to Azure Cosmos DB: Graph API

6/14/2017 • 7 min to read • Edit Online

Azure Cosmos DB is the globally distributed, multi-model database service from Microsoft for mission-critical applications. Azure Cosmos DB provides turn-key global distribution, elastic scaling of throughput and storage worldwide, single-digit millisecond latencies at the 99th percentile, five well-defined consistency levels, and guaranteed high availability, which are all backed by industry-leading SLAs. Azure Cosmos DB automatically indexes data without requiring you to deal with schema and index management. It is multi-model and supports document, key-value, graph, and columnar data models.



The Azure Cosmos DB Graph API provides:

- Graph modeling
- Traversal APIs
- Turn-key global distribution
- Elastic scaling of storage and throughput with less than 10 ms read latencies and less than 15 ms at the 99th percentile
- Automatic indexing with instant query availability
- Tunable consistency levels
- Comprehensive SLAs including 99.99% availability

To query Azure Cosmos DB, you can use the Apache TinkerPop graph traversal language, Gremlin, or other TinkerPop-compatible graph systems like Apache Spark GraphX.

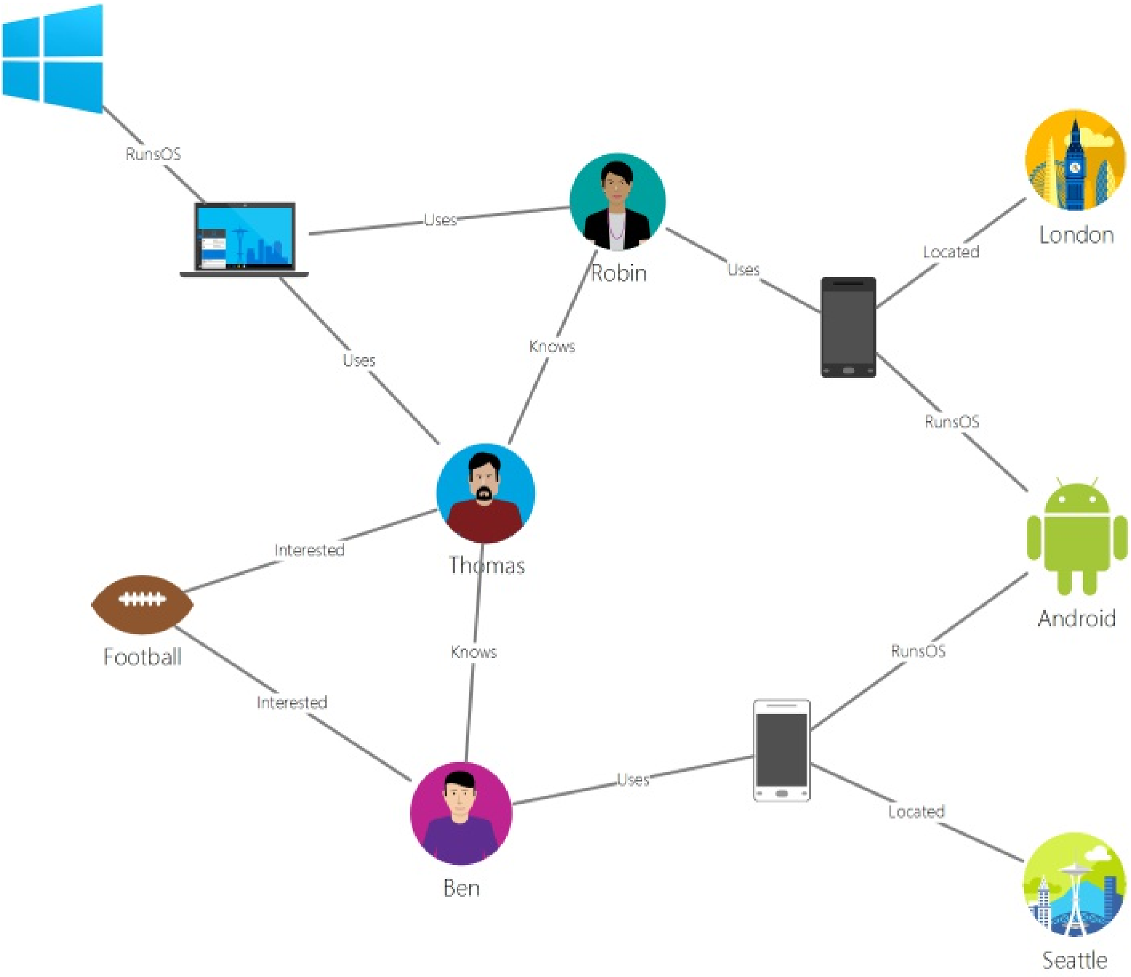
This article provides an overview of the Azure Cosmos DB Graph API and explains how you can use it to store massive graphs with billions of vertices and edges. You can query the graphs with millisecond latency and evolve the graph structure and schema easily.

Graph database

Data as it appears in the real world is naturally connected. Traditional data modeling focuses on entities. For many applications, there is also a need to model or to model both entities and relationships naturally.

A graph is a structure that's composed of vertices and edges. Both vertices and edges can have an arbitrary number of properties. Vertices denote discrete objects such as a person, a place, or an event. Edges denote relationships between vertices. For example, a person might know another person, be involved in an event, and recently been at a location. Properties express information about the vertices and edges. Example properties include a vertex that has a name, age, and edge, which has a timestamp and/or a weight. More formally, this model is known as a property graph. Azure Cosmos DB supports the property graph model.

For example, the following sample graph shows relationships among people, mobile devices, interests, and operating systems.



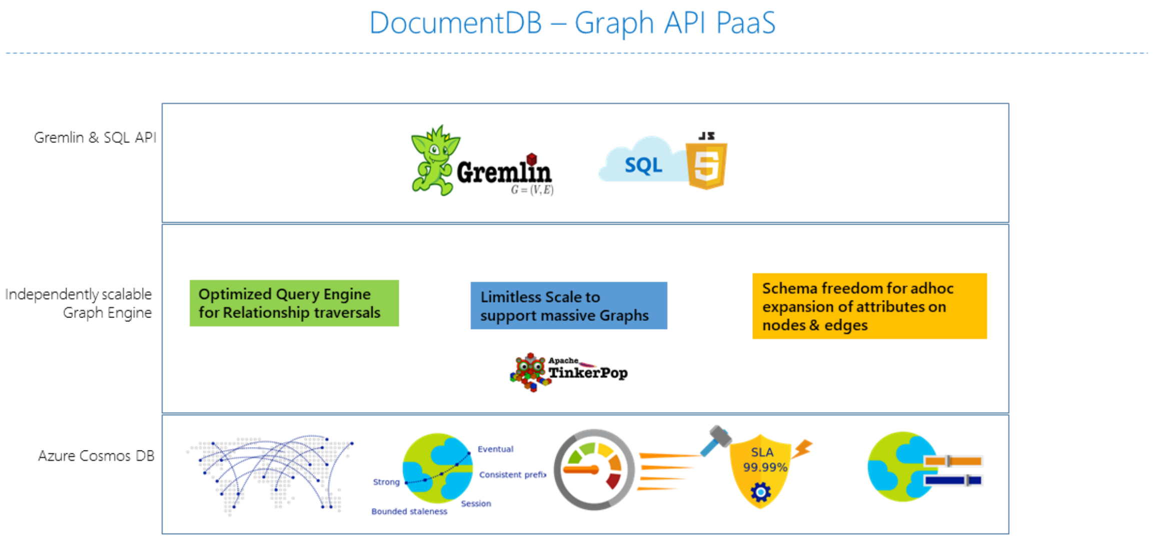
Graphs are useful to understand a wide range of datasets in science, technology, and business. Graph databases let you model and store graphs naturally and efficiently, which makes them useful for many scenarios. Graph databases are typically NoSQL databases because these use cases often also need schema flexibility and rapid iteration.

Graphs offer a novel and powerful data modeling technique. But this fact by itself is not a sufficient reason to use a graph database. For many use cases and patterns that involve graph traversals, graphs outperform traditional SQL and NoSQL databases by orders of magnitude. This difference in performance is further amplified when traversing more than one relationship, like friend-of-a-friend.

You can combine the fast traversals that graph databases provide with graph algorithms, like depth-first search, breadth-first search, and Dijkstra's algorithm, to solve problems in various domains like social networking, content management, geospatial, and recommendations.

Planet-scale graphs with Azure Cosmos DB

Azure Cosmos DB is a fully managed graph database that offers global distribution, elastic scaling of storage and throughput, automatic indexing and query, tunable consistency levels, and support for the TinkerPop standard.


DocumentDB – Graph API PaaS
Gremlin & SQL API
Gremlin
G = (V, E)
SQL
JS
Independently scalable Graph Engine
Optimized Query Engine for Relationship traversals
Limitless Scale to support massive Graphs
Schema freedom for adhoc expansion of attributes on nodes & edges
Apache TinkerPop
Azure Cosmos DB
Eventual
Consistent prefix
Strong
Session
Bounded staleness
SLA 99.99%

Azure Cosmos DB offers the following differentiated capabilities when compared to other graph databases in the market:

- Elastically scalable throughput and storage

  Graphs in the real world need to scale beyond the capacity of a single server. With Azure Cosmos DB, you can scale your graphs seamlessly across multiple servers. You can also scale the throughput of your graph independently based on your access patterns. Azure Cosmos DB supports graph databases that can scale to virtually unlimited storage sizes and provisioned throughput.

- Multi-region replication

  Azure Cosmos DB transparently replicates your graph data to all regions that you've associated with your account. Replication enables you to develop applications that require global access to data. There are tradeoffs in the areas of consistency, availability, and performance and corresponding guarantees. Azure Cosmos DB provides transparent regional failover with multi-homing APIs. You can elastically scale throughput and storage across the globe.

- Fast queries and traversals with familiar Gremlin syntax

  Store heterogeneous vertices and edges and query these documents through a familiar Gremlin syntax. Azure Cosmos DB utilizes a highly concurrent, lock-free, log-structured indexing technology to automatically index all content. This capability enables rich real-time queries and traversals without the need to specify schema hints, secondary indexes, or views. Learn more in Query Graphs using Gremlin.

- Fully managed

  Azure Cosmos DB eliminates 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 graph is automatically backed up and protected against regional failures. You can easily add an Azure Cosmos DB account and provision capacity as you need it so that you can focus on your application instead of operating and managing your database.

- Automatic indexing

  By default, Azure Cosmos DB automatically indexes all the properties within nodes and edges in the graph and does not expect or require any schema or creation of secondary indices.

- Compatibility with Apache TinkerPop

  Azure Cosmos DB natively supports the open-source Apache TinkerPop standard and can be integrated with other TinkerPop-enabled graph systems. So, you can easily migrate from another graph database, like Titan or Neo4j, or use Azure Cosmos DB with graph analytics frameworks like Apache Spark GraphX.

- Tunable consistency levels

  Select from five well-defined consistency levels to achieve optimal trade-off between consistency and performance. For queries and read operations, Azure Cosmos DB offers five distinct consistency levels: strong, bounded-staleness, session, consistent prefix, and eventual. These granular, well-defined consistency levels allow you to make sound tradeoffs among consistency, availability, and latency. Learn more in Using consistency levels to maximize availability and performance in DocumentDB.

Azure Cosmos DB also can use multiple models, like document and graph, within the same containers/databases. You can use a document collection to store graph data side by side with documents. You can use both SQL queries over JSON and Gremlin queries to query the same data as a graph.

Getting started

You can use the Azure command-line interface (CLI), Azure Powershell, or the Azure portal with support for graph API to create Azure Cosmos DB accounts. After you create accounts, the Azure portal provides a service endpoint, like `https://<youraccount>.graphs.azure.com`, that provides a WebSocket front end for Gremlin. You can configure your TinkerPop-compatible tools, like the Gremin Console, to connect to this endpoint and build applications in Java, Node.js, or any Gremlin client driver.

The following table shows popular Gremlin drivers that you can use against Azure Cosmos DB:

DOWNLOAD	DOCUMENTATION
Java	Gremlin JavaDoc
Node.js	Gremlin-JavaScript on Github
Gremlin console	TinkerPop docs

Azure Cosmos DB also provides a .NET library that has Gremlin extension methods on top of the Azure Cosmos DB SDKs via NuGet. This library provides an "in-process" Gremlin server that you can use to connect directly to DocumentDB data partitions.

DOWNLOAD	DOCUMENTATION
.NET	Microsoft.Azure.Graphs

By using the Azure Cosmos DB Emulator, you can use the Graph API to develop and test locally without creating an Azure subscription or incurring any costs. When you're satisfied with how your application is working in the Emulator, you can switch to using an Azure Cosmos DB account in the cloud.

Scenarios for graph support of Azure Cosmos DB

Here are some scenarios where graph support of Azure Cosmos DB can be used:

- Social networks

  By combining data about your customers and their interactions with other people, you can develop personalized experiences, predict customer behavior, or connect people with others with similar interests.

Azure Cosmos DB can be used to manage social networks and track customer preferences and data.

- Recommendation engines

  This scenario is commonly used in the retail industry. By combining information about products, users, and user interactions, like purchasing, browsing, or rating an item, you can build customized recommendations. The low latency, elastic scale, and native graph support of Azure Cosmos DB is ideal for modeling these interactions.

- Geospatial

  Many applications in telecommunications, logistics, and travel planning need to find a location of interest within an area or locate the shortest/optimal route between two locations. Azure Cosmos DB is a natural fit for these problems.

- Internet of Things

  With the network and connections between IoT devices modeled as a graph, you can build a better understanding of the state of your devices and assets and learn how changes in one part of the network can potentially affect another part.

Next steps

To learn more about graph support in Azure Cosmos DB, see:

- Get started with the Azure Cosmos DB graph tutorial.
- Learn about how to query graphs in Azure Cosmos DB using Gremlin.

Azure Cosmos DB database security

5/30/2017 • 7 min to read • Edit Online

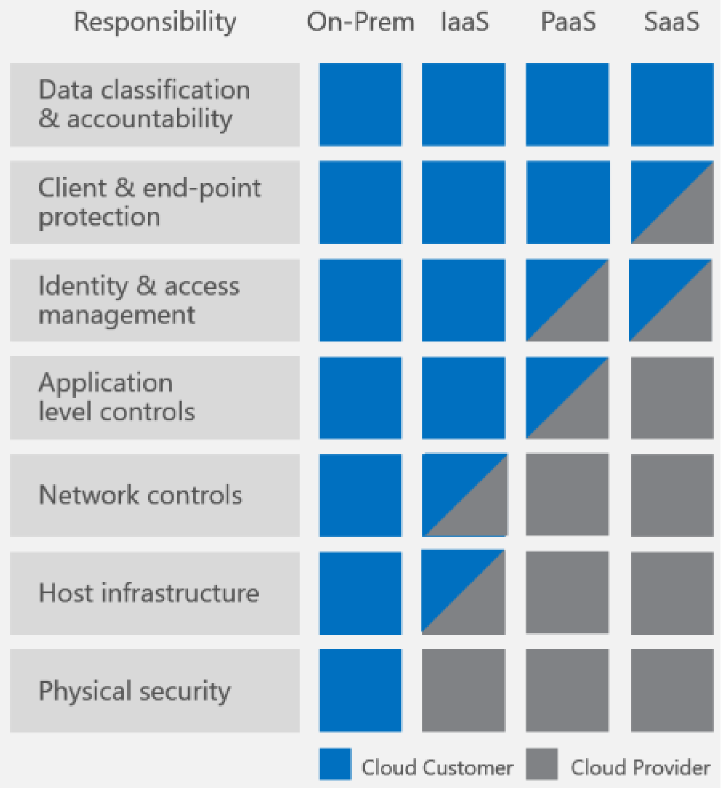
This article discusses database security best practices and key features offered by Azure Cosmos DB to help you prevent, detect, and respond to database breaches.

What's new in Azure Cosmos DB security?

Encryption at rest is now available for documents stored in Azure Cosmos DB in all Azure regions except government regions, Azure in China, and Azure in Germany. The remaining regions will be enabled next week, along with encryption at rest on backups. Encryption at rest is applied automatically for both new and existing customers in these regions. There is no need to configure anything; and you get the same great latency, throughput, availability, and functionality as before with the benefit of knowing your data is safe and secure with encryption at rest.

How do I secure my database?

Data security is a shared responsibility between you, the customer, and your database provider. Depending on the database provider you choose, the amount of responsibility you carry can vary. If you choose an on-premises solution, you need to provide everything from end-point protection to physical security of your hardware - which is no easy task. If you choose a PaaS cloud database provider such as Azure Cosmos DB, your area of concern shrinks considerably. The following image, borrowed from Microsoft's Shared Responsibilities for Cloud Computing white paper, shows how your responsibility decreases with a PaaS provider like Azure Cosmos DB.



Responsibility	On-Prem	IaaS	PaaS	SaaS
Data classification & accountability	🟦	🟦	🟦	🟦
Client & end-point protection	🟦	🟦	🟦	◩
Identity & access management	🟦	🟦	◩	◩
Application level controls	🟦	🟦	◩	⬛
Network controls	🟦	◩	⬛	⬛
Host infrastructure	🟦	◩	⬛	⬛
Physical security	🟦	⬛	⬛	⬛

🟦 Cloud Customer ⬛ Cloud Provider

The diagram above shows high-level cloud security components, but what items do you need to worry about specifically for your database solution? And how can you compare solutions to each other?

We recommend the following checklist of requirements on which to compare database systems:

- Network security and firewall settings

- User authentication and fine grained user controls
- Ability to replicate data globally for regional failures
- Ability to perform failovers from one data center to another
- Local data replication within a data center
- Automatic data backups
- Restoration of deleted data from backups
- Protect and isolate sensitive data
- Monitoring for attacks
- Responding to attacks
- Ability to geo-fence data to adhere to data governance restrictions
- Physical protection of servers in protected data centers

And although it may seem obvious, recent large-scale database breaches remind us of the simple but critical importance of the following requirements:

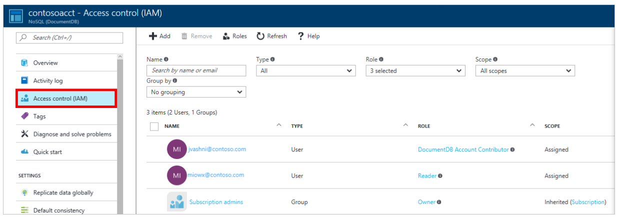
- Patched servers that are kept up to date
- HTTPS by default/SSL encryption
- Administrative accounts with strong passwords

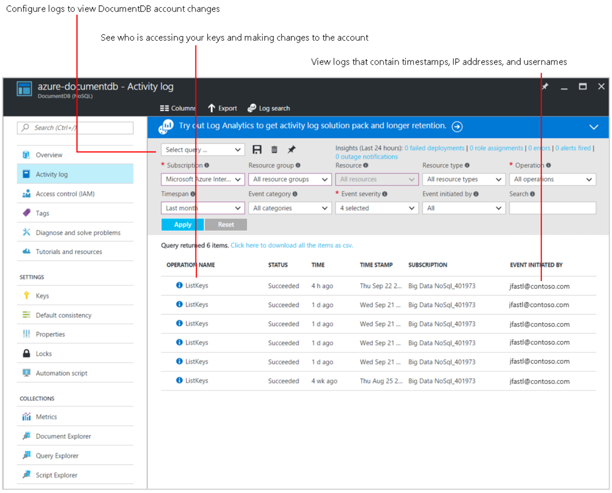
How does Azure Cosmos DB secure my database?

Let's look back at the preceding list - how many of those security requirements does Azure Cosmos DB provide? Every single one.

Let's dig into each one in detail.

SECURITY REQUIREMENT	AZURE COSMOS DB'S SECURITY APPROACH
Network security	Using an IP firewall is the first layer of protection to secure your database. Azure Cosmos DB supports policy driven IP-based access controls for inbound firewall support. The IP-based access controls are similar to the firewall rules used by traditional database systems, but they are expanded so that an Azure Cosmos DB database account is only accessible from an approved set of machines or cloud services. Azure Cosmos DB enables you to enable a specific IP address (168.61.48.0), an IP range (168.61.48.0/8), and combinations of IPs and ranges. All requests originating from machines outside this allowed list are blocked by Azure Cosmos DB. Requests from approved machines and cloud services then must complete the authentication process to be given access control to the resources. Learn more in Azure Cosmos DB firewall support.

SECURITY REQUIREMENT	AZURE COSMOS DB'S SECURITY APPROACH
Authorization	Azure Cosmos DB uses hash-based message authentication code (HMAC) for authorization. Each request is hashed using the secret account key, and the subsequent base-64 encoded hash is sent with each call to Azure Cosmos DB. To validate the request, the Azure Cosmos DB service uses the correct secret key and properties to generate a hash, then it compares the value with the one in the request. If the two values match, the operation is authorized successfully and the request is processed, otherwise there is an authorization failure and the request is rejected. You can use either a master key, or a resource token allowing fine-grained access to a resource such as a document. Learn more in Securing access to Azure Cosmos DB resources.
Users and permissions	Using the master key for the account, you can create user resources and permission resources per database. A resource token is associated with a permission in a database and determines whether the user has access (read-write, read-only, or no access) to an application resource in the database. Application resources include collections, documents, attachments, stored procedures, triggers, and UDFs. The resource token is then used during authentication to provide or deny access to the resource. Learn more in Securing access to Azure Cosmos DB resources.
Active directory integration (RBAC)	You can also provide access to the database account using Access control (IAM) in the Azure portal. IAM provides role-based access control and integrates with Active Directory. You can use built in roles or custom roles for individuals and groups as shown in the following image.  contosoacct - Access control (IAM) NoSQL (DocumentDB) Search (Ctrl+/) Overview Activity log Access control (IAM) Tags Diagnose and solve problems Quick start SETTINGS Replicate data globally Default consistency Add Remove Roles Refresh Help Name: Search by name or email Type: All Role: 3 selected Scope: All scopes Group by: No grouping 3 items (2 Users, 1 Groups) NAME / TYPE / ROLE / SCOPE jvashni@contoso.com / User / DocumentDB Account Contributor / Assigned miowx@contoso.com / User / Reader / Assigned Subscription admins / Group / Owner / Inherited (Subscription)
Global replication	Azure Cosmos DB offers turnkey global distribution, which enables you to replicate your data to any one of Azure's world-wide datacenters with the click of a button. Global replication lets you scale globally and provide low-latency access to your data around the world. In the context of security, global replication insures data protection against regional failures. Learn more in Distribute data globally.

SECURITY REQUIREMENT	AZURE COSMOS DB'S SECURITY APPROACH
Regional failovers	If you have replicated your data in more than one data center, Azure Cosmos DB automatically rolls over your operations should a regional data center go offline. You can create a prioritized list of failover regions using the regions in which your data is replicated. Learn more in Regional Failovers in Azure Cosmos DB.
Local replication	Even within a single data center, Azure Cosmos DB automatically replicates data for high availability giving you the choice of consistency levels. This guarantees a 99.99% uptime availability SLA and comes with a financial guarantee - something no other database service can provide.
Automated online backups	Azure Cosmos DB databases are backed up regularly and stored in a georedundant store. Learn more in Automatic online backup and restore with Azure Cosmos DB.
Restore deleted data	The automated online backups can be used to recover data you may have accidentally deleted up to ~30 days after the event. Learn more in Automatic online backup and restore with Azure Cosmos DB
Protect and isolate sensitive data	All data in the regions listed in What's new? is now encrypted at rest. PII and other confidential data can be isolated to specific collections and read-write, or read-only access can be limited to specific users.
Monitor for attacks	By using audit logging and activity logs, you can monitor your account for normal and abnormal activity. You can view what operations were performed on your resources, who initiated the operation, when the operation occurred, the status of the operation, and much more.  Configure logs to view DocumentDB account changes See who is accessing your keys and making changes to the account View logs that contain timestamps, IP addresses, and usernames

azure-documentdb - Activity log
DocumentDB (NoSQL)

Try out Log Analytics to get activity log solution pack and longer retention.

Select query ...

Insights (Last 24 hours): 0 failed deployments | 0 role assignments | 0 errors | 0 alerts fired | 0 outage notifications

* Subscription	Resource group	Resource	Resource type	* Operation
Microsoft Azure Inter...	All resource groups	All resources	All resource types	All operations

Timespan	Event category	* Event severity	Event initiated by	Search
Last month	All categories	4 selected	All	

Query returned 6 items. Click here to download all the items as csv.

OPERATION NAME	STATUS	TIME	TIME STAMP	SUBSCRIPTION	EVENT INITIATED BY
ListKeys	Succeeded	4 h ago	Thu Sep 22 2...	Big Data NoSql_401973	jfastl@contoso.com
ListKeys	Succeeded	1 d ago	Wed Sep 21 ...	Big Data NoSql_401973	jfastl@contoso.com
ListKeys	Succeeded	1 d ago	Wed Sep 21 ...	Big Data NoSql_401973	jfastl@contoso.com
ListKeys	Succeeded	1 d ago	Wed Sep 21 ...	Big Data NoSql_401973	jfastl@contoso.com
ListKeys	Succeeded	1 d ago	Wed Sep 21 ...	Big Data NoSql_401973	jfastl@contoso.com
ListKeys	Succeeded	4 wk ago	Thu Aug 25 2...	Big Data NoSql_401973	jfastl@contoso.com

SECURITY REQUIREMENT	AZURE COSMOS DB'S SECURITY APPROACH
Respond to attacks	Once you have contacted Azure support to report a potential attack, a 5-step incident response process is kicked off. The goal of the 5-step process is to restore normal service security and operations as quickly as possible after an issue is detected and an investigation is started. Learn more in Microsoft Azure Security Response in the Cloud.
Geo-fencing	Azure Cosmos DB ensures data governance and compliance for sovereign regions (for example, Germany, China, US Gov).
Protected facilities	Data in Azure Cosmos DB is stored on SSDs in Azure's protected data centers. Learn more in Microsoft global datacenters
HTTPS/SSL/TLS encryption	All client-to-service Azure Cosmos DB interactions are SSL/TLS 1.2 enforced. Also, all intra datacenter and cross datacenter replication is SSL/TLS 1.2 enforced.
Encryption at rest	All data stored into Azure Cosmos DB is encrypted at rest. Learn more in Azure Cosmos DB encryption at rest
Patched servers	As a managed database, Azure Cosmos DB eliminates the need to manage and patch servers, that's done for you, automatically.
Administrative accounts with strong passwords	It's hard to believe we even need to mention this requirement, but unlike some of our competitors, it's impossible to have an administrative account with no password in Azure Cosmos DB. Security via SSL and HMAC secret based authentication is baked in by default.
Security and data protection certifications	Azure Cosmos DB has ISO 27001, European Model Clauses (EUMC), and HIPAA certifications. Additional certifications are in progress.

Next steps

For more details about master keys and resource tokens, see Securing access to Azure Cosmos DB data.

For more details about Microsoft certifications, see Azure Trust Center.

Common Azure Cosmos DB use cases

5/30/2017 • 9 min to read • Edit Online

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

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

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

Introduction

Azure Cosmos DB is Microsoft's globally distributed database service. The service is designed to allow customers to elastically (and independently) scale throughput and storage across any number of geographical regions. Cosmos DB is the first globally distributed database service in the market today to offer comprehensive service level agreements encompassing throughput, latency, availability, and consistency.

The Cosmos DB project started in 2011 as "Project Florence" to address developer pain-points that are faced by large Internet-scale applications inside Microsoft. Observing that these problems are not unique to Microsoft's applications, we decided to make Cosmos DB generally available to external developers in 2015 in the form of Azure DocumentDB. The service is used ubiquitously internally within Microsoft, and is one of the fastest-growing services used by Azure developers externally.

Azure Cosmos DB is a global distributed, multi-model 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 and globally. It supports multiple data models (key-value, documents, graphs and columnar) and many APIs for data access including MongoDB, DocumentDB SQL, Gremlin, and Azure Tables natively, and in an extensible manner.

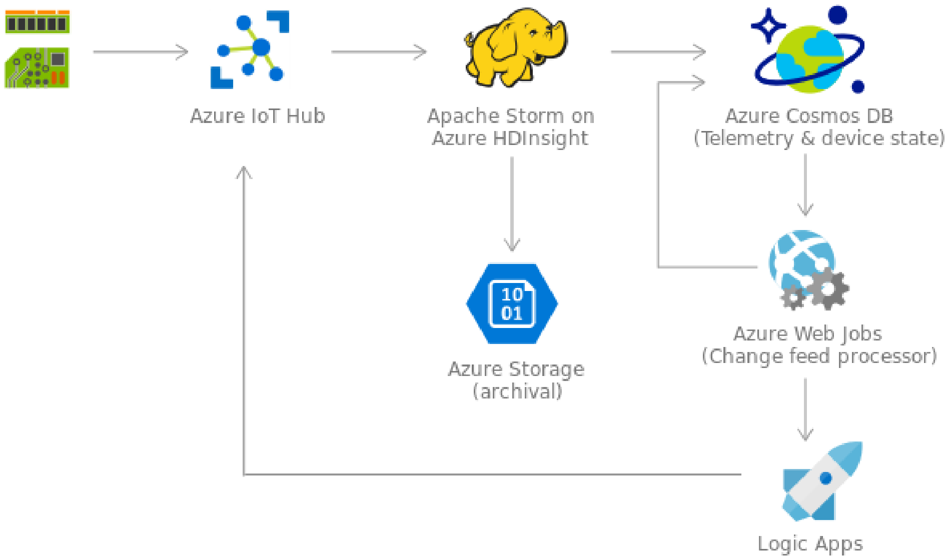
The following are some attributes of Cosmos DB that make it well-suited for high-performance applications with global ambition.

- Cosmos DB natively partitions your data for high availability and scalability. Cosmos DB offers 99.99% guarantees for availability, throughput, low latency, and consistency.
- Cosmos DB has SSD backed storage with low-latency order-of-millisecond response times.
- Cosmos DB's support for consistency levels like eventual, consistent prefix, session, and bounded-staleness allows for full flexibility and low cost-to performance-ratio. No database service offers as much flexibility as Cosmos DB in levels consistency.
- Cosmos DB has a flexible data-friendly pricing model that meters storage and throughput independently.
- Cosmos DB's reserved throughput model allows you to think in terms of number of reads/writes instead of CPU/memory/IOPs of the underlying hardware.
- Cosmos DB's design lets you scale to massive request volumes in the order of trillions of requests per day.

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

IoT and telematics

IoT use cases commonly share some patterns in how they ingest, process, and store data. First, these systems need to ingest bursts of data from device sensors of various locales. Next, these systems process and analyze streaming data to derive real-time insights. The data is then archived to cold storage for batch analytics. Microsoft Azure offers rich services that can be applied for IoT use cases including Azure Cosmos DB, Azure Event Hubs, Azure Stream Analytics, Azure Notification Hub, Azure Machine Learning, Azure HDInsight, and PowerBI.


Azure IoT Hub
Apache Storm on Azure HDInsight
Azure Cosmos DB (Telemetry & device state)
10 01
Azure Storage (archival)
Azure Web Jobs (Change feed processor)
Logic Apps

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 Cosmos DB for adhoc querying. Once the data is loaded into Cosmos DB, the data is ready to be queried. The data in Cosmos DB can be used as reference data as part of real-time analytics. In addition, data can further be refined and processed by connecting Cosmos DB data to HDInsight for Pig, Hive or Map/Reduce jobs. Refined data is then loaded back to Cosmos DB for reporting.

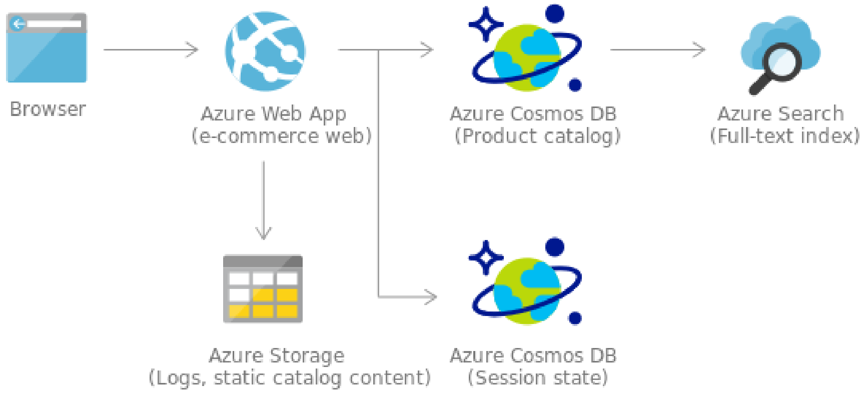
For a sample IoT solution using Cosmos DB, EventHubs and Storm, see the hdinsight-storm-examples repository on GitHub.

For more information on Azure offerings for IoT, see Create the Internet of Your Things.

Retail and marketing

Cosmos DB is used extensively in Microsoft's own e-commerce platforms, that run the Windows Store and XBox Live. It is also 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. Cosmos DB supports flexible schemas and hierarchical data, and thus it is well suited for storing product catalog data.


Browser
Azure Web App (e-commerce web)
Azure Cosmos DB (Product catalog)
Azure Search (Full-text index)
Azure Storage (Logs, static catalog content)
Azure Cosmos DB (Session state)

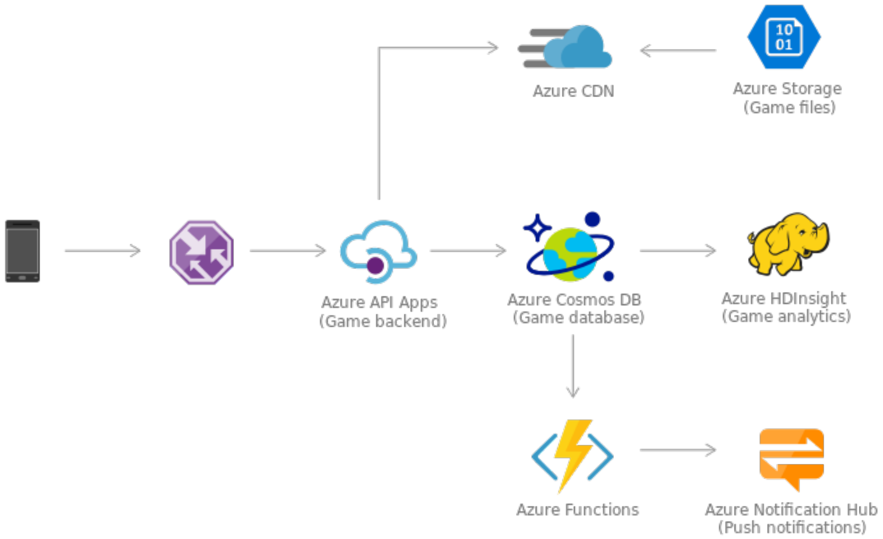
In addition, data stored in Cosmos DB can be integrated with HDInsight for big data analytics via Pig, Hive, or Map/Reduce jobs. For details on the Hadoop Connector for Cosmos DB, see Run a Hadoop job with Cosmos DB and HDInsight.

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 often require single-millisecond latencies for reads and writes to provide an engaging in-game experience. A game database needs to be fast and be able to handle massive spikes in request rates during new game launches and feature updates.

Cosmos DB is used by games like The Walking Dead: No Man's Land by Next Games, and Halo 5: Guardians. Cosmos DB provides the following benefits to game developers:

- Cosmos DB 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.
- Cosmos DB supports millisecond reads and writes to help avoid any lags during game play.
- Cosmos DB'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.
- Cosmos DB as a managed platform-as-a-service (PaaS) required minimal setup and management work to allow for rapid iteration, and reduce time to market.



Web and mobile applications

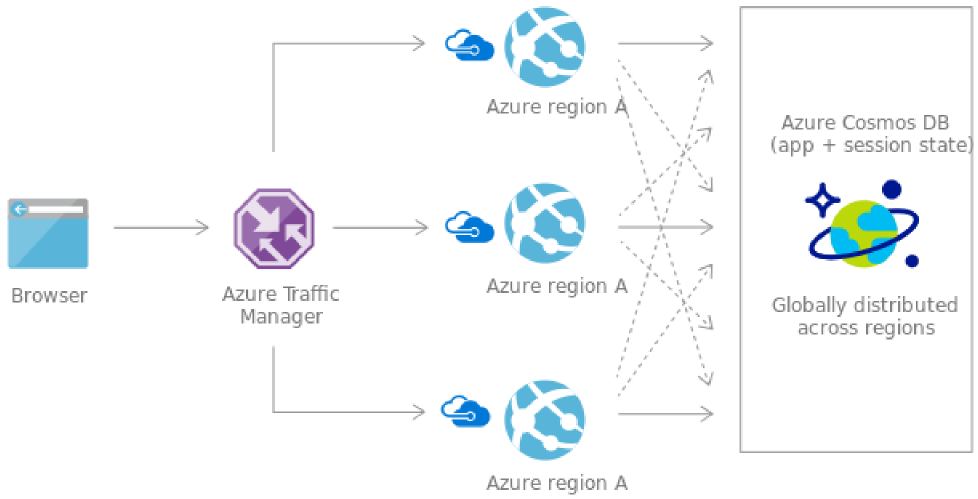
Cosmos DB is commonly used within web and mobile applications, and is particularly well suited for modeling social interactions, integrating with third-party services, and for building rich personalized experiences. The Cosmos DB SDKs can be used build rich iOS and Android applications using the popular Xamarin framework.

Social Applications

A common use case for Cosmos DB 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 Cosmos DB 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 Cosmos DB, data is ready to be queried at any time. Hence, these applications have the flexibility to retrieve projections as per their respective needs.

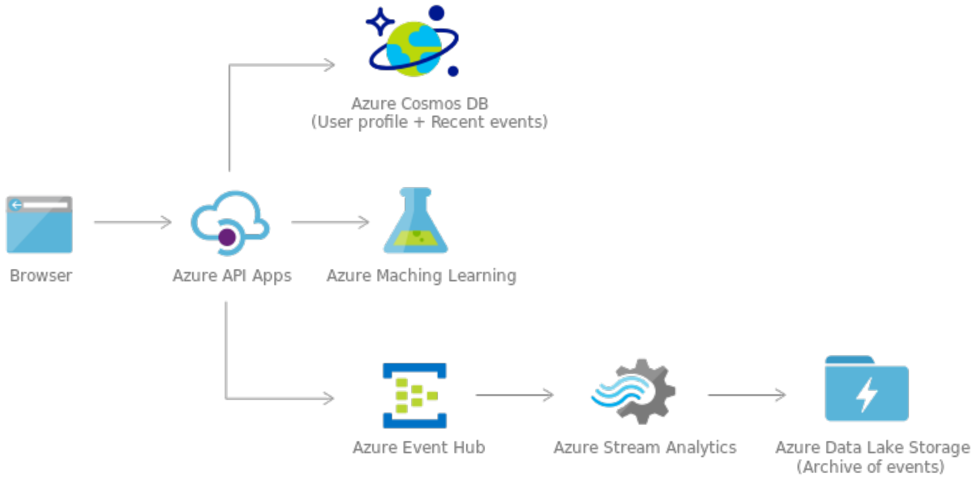
Many of the social applications run at global scale and can exhibit unpredictable usage patterns. Flexibility in scaling the data store is essential as the application layer scales to match usage demand. You can scale out by adding additional data partitions under a Cosmos DB account. In addition, you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


Browser
Azure Traffic Manager
Azure region A
Azure region A
Azure region A
Azure Cosmos DB (app + session state)
Globally distributed across 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 to render UI elements and experiences quickly.

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


Azure Cosmos DB (User profile + Recent events)
Browser
Azure API Apps
Azure Maching Learning
Azure Event Hub
Azure Stream Analytics
Azure Data Lake Storage (Archive of events)

Next steps

To get started with Azure Cosmos DB, follow our quick starts which will walk you through creating an account and getting started with Cosmos DB.

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

- Jet.com. E-commerce challenger eyes the top spot, runs on the Microsoft cloud, leverages Cosmos DB at a global scale.

- Asos.com. Asos.com is a British online fashion and beauty store. Primarily aimed at young adults, Asos sells over 850 brands as well as its own range of clothing and accessories.
- Toyota. Toyota Motor Corporation is a Japanese automotive manufacturer. Toyota leveraged Cosmos DB for a global IoT app.
- Citrix. Citrix develops single-sign-on solution using Azure Service Fabric and Azure Cosmos DB
- TEXA TEXA's revolutionary IoT solution for vehicle owners helps save time, money, gas—and possibly lives.
- Domino's Pizza. Domino's Pizza Inc. is an American pizza restaurant chain.
- Johnson Controls. Johnson Controls is a global diversified technology and multi industrial leader serving a wide range of customers in more than 150 countries.
- Microsoft Windows, Universal Store, Azure IoT Hub, Xbox Live, and other Internet-scale services. How Microsoft builds massively scalable services using Azure DocumentDB.
- Microsoft Data and Analytics team. Microsoft's Data and Analytics team achieves planet-scale big-data collection with Azure Cosmos DB
- Sulekha.com. Sulekha uses Azure Cosmos DB to connect customers and businesses across India .
- NewOrbit. NewOrbit takes flight with Azure Cosmos DB.
- Affinio. Affinio switches from AWS to Azure Cosmos DB to harness social data at scale.
- Next Games. The Walking Dead: No Man's Land game soars to #1 supported by Azure Cosmos DB.
- Halo. How Halo 5 implemented social gameplay using Azure Cosmos DB.
- Cortana Analytics Gallery. Cortana Analytics Gallery - a scalable community site built on Azure Cosmos DB.
- 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 Azure Cosmos DB

5/30/2017 • 13 min to read • Edit Online

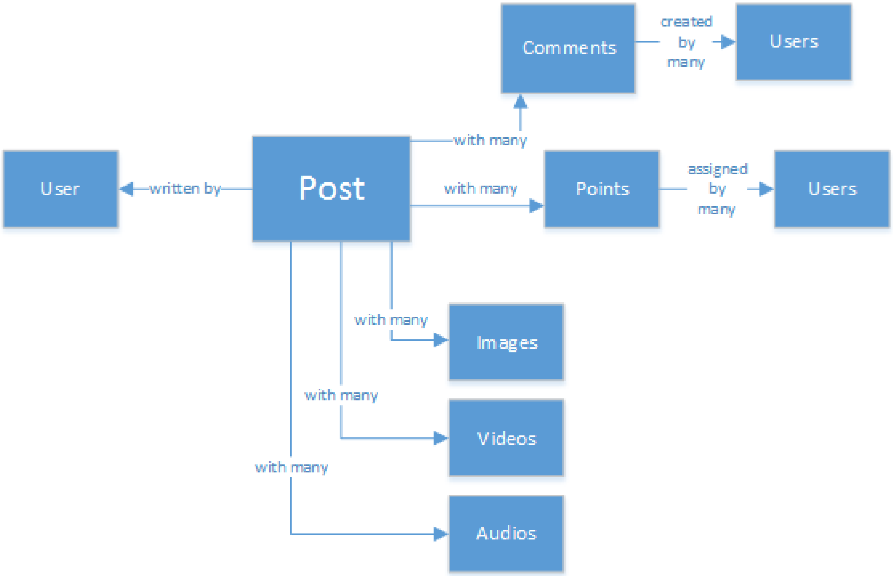
Living in a massively-interconnected society means that, at some point in life, you become part of a **social network**. We use social networks to keep in touch with friends, colleagues, family, or sometimes to share our passion with people with common interests.

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

This article will guide you into modeling your social platform's data with Azure's NoSQL database Azure Cosmos

DB in a cost-effective way while leveraging other Azure Cosmos DB features like the Gremlin Graph API. 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 Cosmos DB 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, Cosmos DB 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 Cosmos DB 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. Cosmos DB has a maximum document size limit, and reading/writing large documents can impact the scalability of your application. 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 lead to a large document size, and might 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 using Azure Cosmos DB Gremlin Graph API, to create vertexes for each user and edges that maintain the "A-follows-B" relationships. The Graph API let's you not only obtain the followers of a certain user but create more complex queries to even suggest people in common. If we add to the graph the Content Categories that people like or enjoy, we can start weaving experiences that include smart content discovery, suggesting content that those we follow like, or finding people with whom we might have much in common.

The User Statistics document can still be used to create cards in the UI or quick profile previews.

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.

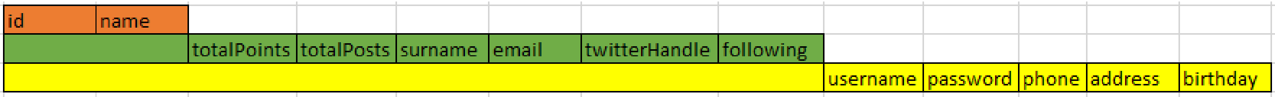
In order to allow for faster queries, we incur data duplication. The problem with this side-effect is that if by some action, a user’s data changes, we need to find all the activities he ever did and update them all. Doesn’t sound very practical, right?

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 Cosmos DB, 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 Cosmos DB, in Azure SQL Database or Azure Storage Tables.

Why would we split the user and even store this information in different places? Because 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 Cosmos DB, 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 automagically add, update or remove your objects in the indexes. They support an Azure SQL Database indexers, Azure Blobs indexers and thankfully, Azure Cosmos DB indexers. The transition of information from Cosmos DB 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.

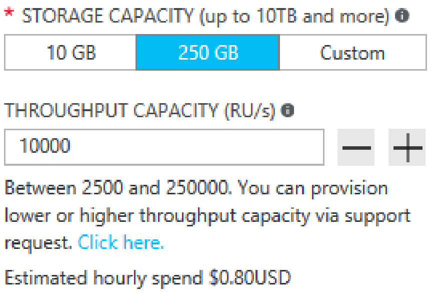
A planet-scale social experience

There is a last, but not least, important topic I must address: **scalability**. When designing an architecture it's crucial that each component can scale on its own, either because we need to process more data or because we want to have a bigger geographical coverage (or both!). Thankfully, achieving such a complex task is a **turnkey experience** with Cosmos DB.

Cosmos DB supports dynamic partitioning out-of-the-box by automatically creating partitions based on a given **partition key** (defined as one of the attributes in your documents). Defining the correct partition key must be done at design time and keeping in mind the best practices available; in the case of a social experience, your partitioning strategy must be aligned with the way you query (reads within the same partition are desirable) and write (avoid "hot spots" by spreading writes on multiple partitions). Some options are: partitions based on a temporal key (day/month/week), by content category, by geographical region, by user; it all really depends on how you will query the data and show it in your social experience.

One interesting point worth mentioning is that Cosmos DB will run your queries (including aggregates) across all your partitions transparently, you don't need to add any logic as your data grows.

With time, you will eventually grow in traffic and your resource consumption (measured in RUs, or Request Units) will increase. You will read and write more frequently as your userbase grows and they will start creating and reading more content; the ability of **scaling your throughput** is vital. Increasing our RUs is very easy, we can do it with a few clicks on the Azure Portal or by issuing commands through the API.


* STORAGE CAPACITY (up to 10TB and more)
10 GB
250 GB
Custom
THROUGHPUT CAPACITY (RU/s)
—
+

10000

Between 2500 and 250000. You can provision lower or higher throughput capacity via support request. Click here.

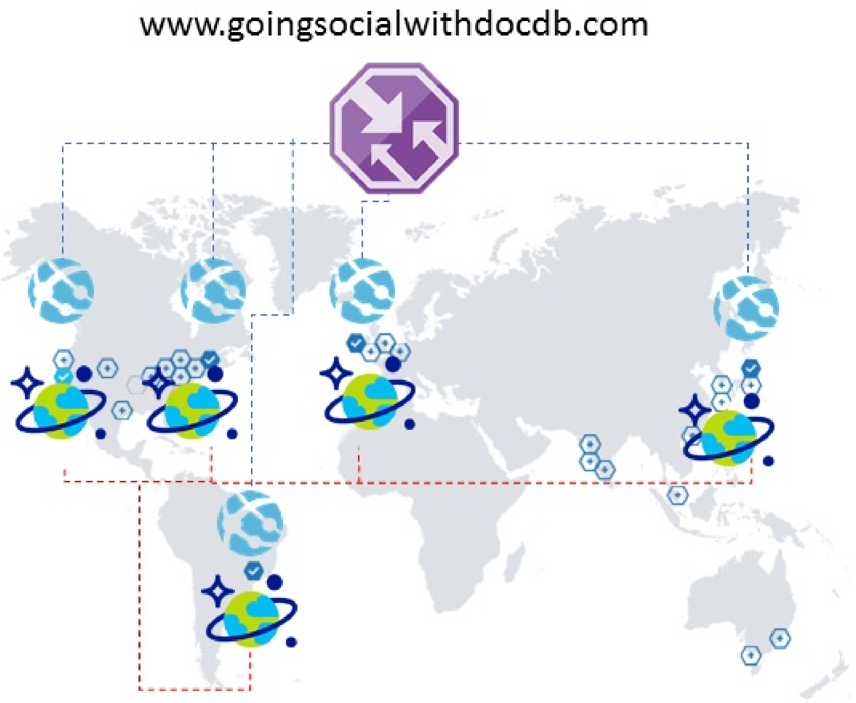
Estimated hourly spend $0.80USD

What happens if things keep getting better and users from another region, country or continent, notice your platform and start using it, what a great surprise!

But wait... you soon realize their experience with your platform is not optimal; they are so far away from your operational region that the latency is terrible, and you obviously don't want them to quit. If only there was an easy way of **extending your global reach**... but there is!

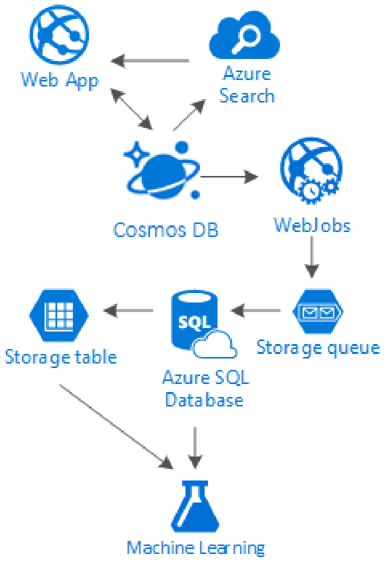
Cosmos DB lets you replicate your data globally and transparently with a couple of clicks and automatically select among the available regions from your client code. This also means that you can have multiple failover regions.

When you replicate your data globally, you need to make sure that your clients can take advantage of it. If you are using a web frontend or accesing APIs from mobile clients, you can deploy Azure Traffic Manager and clone your Azure App Service on all the desired regions, using a Performance configuration to support your extended global coverage. When your clients access your frontend or APIs, they will be routed to the closest App Service, which in turn, will connect to the local Cosmos DB replica.



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 Cosmos DB 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

To learn more about use cases for Cosmos DB, see Common Cosmos DB use cases.

Azure Cosmos DB hierarchical resource model and core concepts

5/30/2017 • 24 min to read • Edit Online

The database entities that Azure Cosmos DB 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.

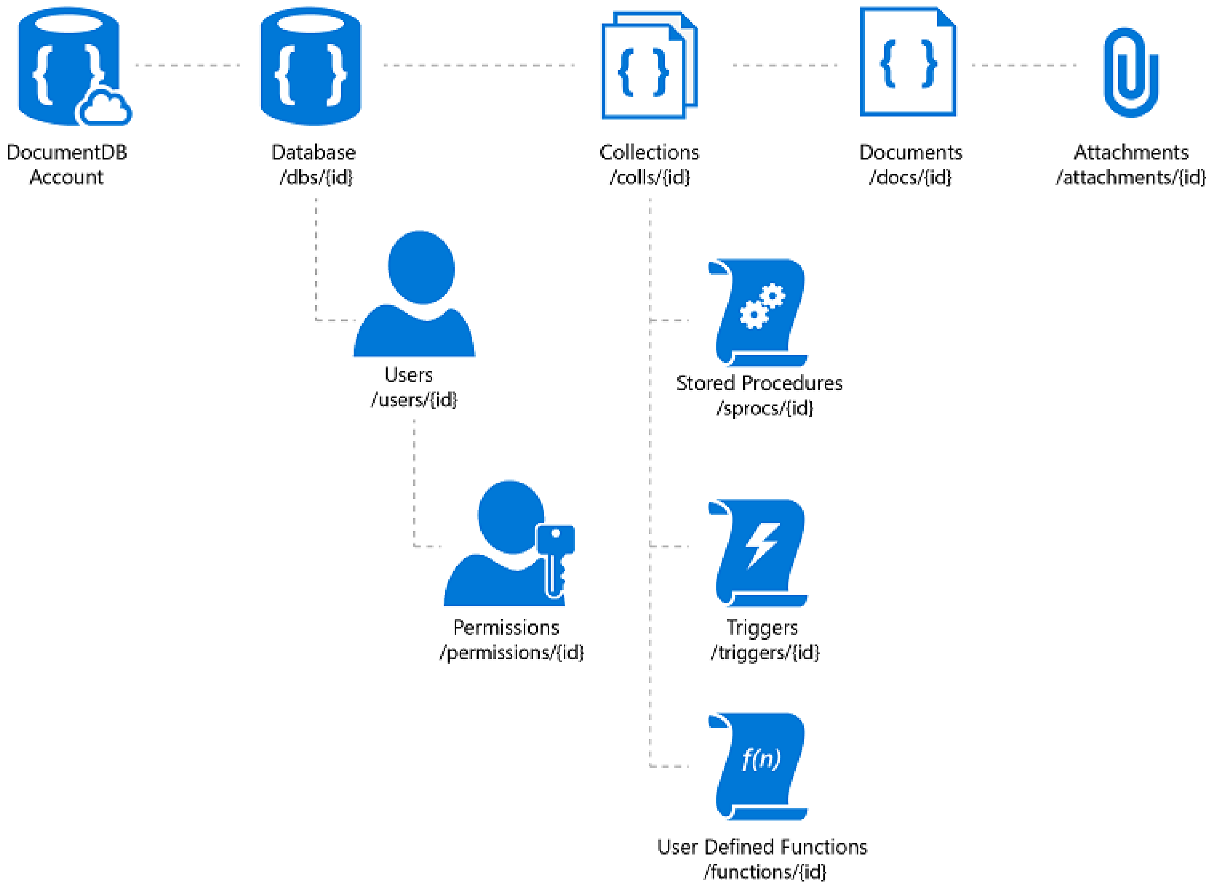
By reading this article, you'll be able to answer the following questions:

- What is Cosmos DB'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 Cosmos DB 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
>
> Cosmos DB 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 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**. 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. 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 API 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.
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 Cosmos DB 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 Cosmos DB, 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

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

Each resource has a unique user defined name exposed via the id property. 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 Cosmos DB 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 Cosmos DB database accounts using your Azure subscription.

You can create and manage Cosmos DB database accounts 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

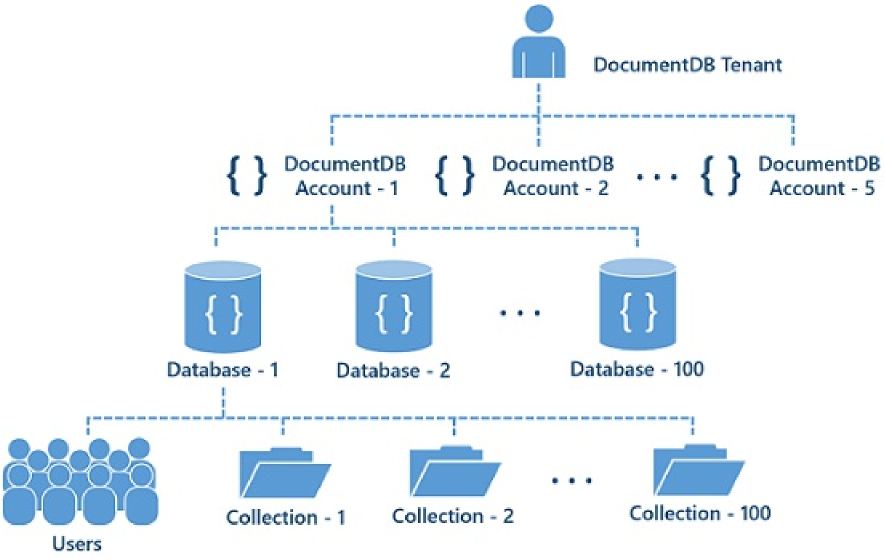
As part of provisioning and managing a database account you can configure and read the following properties:

Property Name	Description
Consistency Policy	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. Note that this property only applies to the *user defined resources*. All system defined resources are configured to support reads/queries with strong consistency.
Authorization Keys	These are the primary and secondary master and readonly keys that provide administrative access to all of the resources under the database account.

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

Databases

A Cosmos DB 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 Cosmos DB 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 within collections.

Elastic scale of a Cosmos DB database

A Cosmos DB 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 Cosmos DB is not scoped to a single machine. With Cosmos DB, 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 Cosmos DB at a consumer scale by creating extremely large Cosmos DB 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 Cosmos DB 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 Cosmos DB resource model, databases can be created, replaced, deleted, read or enumerated easily using either Azure Cosmos DB REST APIs or any of the client SDKs. Cosmos DB 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 Cosmos DB collection is a container for your JSON documents.

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 Cosmos DB based on the storage size and the provisioned throughput of your collection. Every partition in Cosmos DB has a fixed amount of SSD-backed storage associated with it, and is replicated for high availability. Partition management is fully managed by Azure Cosmos DB, and you do not have to write complex code or manage your partitions. Cosmos DB collections are **practically unlimited** in terms of storage and throughput.

Automatic indexing of collections

Cosmos DB 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, Cosmos DB 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 Cosmos DB and is enabled by write-optimized, lock-free and log-structured index maintenance techniques. Cosmos DB 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 Cosmos DB 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 API 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 API 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 API's automatic indexing techniques as well as DocumentDB API's SQL dialect. The DocumetDB API 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 Cosmos DB query model attempts to strike a balance between functionality, efficiency and simplicity. The Cosmos DB database engine natively compiles and executes the SQL query statements. You can query a collection using the Azure Cosmos DB REST APIs or any of the client SDKs. The .NET SDK comes with a LINQ provider.

> TIP
>
> You can try out DocumentDB API 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, Cosmos DB 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. Cosmos DB 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, Cosmos DB 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 API 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 Azure Cosmos DB 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 API can be created, deleted, read or enumerated easily using either the DocumentDB API REST APIs or any of the client SDKs. DocumentDB API 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, Cosmos DB 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 Cosmos DB JavaScript SDK to register, execute, and unregister a stored procedure, trigger, and a UDF. The JavaScript SDK is a simple wrapper over the Cosmos DB 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 API.

```
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 API 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. Cosmos DB does not mandate any schema and does not require secondary indexes in order to support querying over documents in a collection. The maximum size for a document is 2 MB.

Being a truly open database service, Cosmos DB does not invent any specialized data types (e.g. date time) or specific encodings for JSON documents. Note that Cosmos DB does not require any special JSON conventions to codify the relationships among various documents; the SQL syntax of Cosmos DB 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

Cosmos DB allows you to store binary blobs/media either with Cosmos DB (maximum of 2 GB per account) 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 Cosmos DB 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 Cosmos DB 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 Cosmos DB 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 Cosmos DB, the _media property of the attachment will reference the media by its URI. Cosmos DB will ensure to garbage collect the media when all of the outstanding references are dropped. Cosmos DB 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

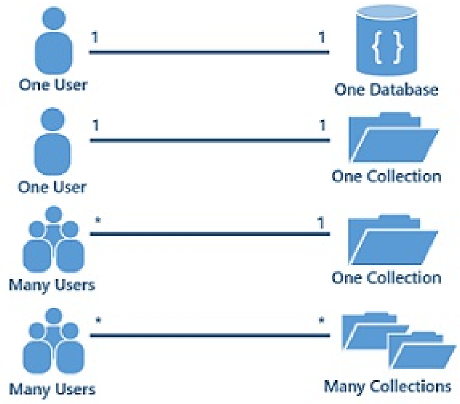
A Cosmos DB user represents a logical namespace for grouping permissions. A Cosmos DB user may correspond to a user in an identity management system or a predefined application role. For Cosmos DB, a user simply represents an abstraction to group a set of permissions under a database.

For implementing multi-tenancy in your application, you can create users in Cosmos DB 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 Cosmos DB database and associate fine grained permissions to each user.



Sharding strategies and modeling users

Like all other resources, users in Cosmos DB can be created, replaced, deleted, read or enumerated easily using either REST APIs or any of the client SDKs. Cosmos DB 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 Cosmos DB 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 Cosmos DB can be created, replaced, deleted, read or enumerated easily using either REST APIs or any of the client SDKs. Cosmos DB 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 Cosmos DB resources.

SQL query and SQL syntax in Azure Cosmos DB

6/9/2017 • 51 min to read • Edit Online

Microsoft Azure Cosmos DB supports querying documents using SQL (Structured Query Language) as a JSON query language. Cosmos DB 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 Cosmos DB 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. Cosmos DB 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 API 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 Cosmos DB's querying capabilities, and by visiting our Query Playground, where you can try out Cosmos DB 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 Cosmos DB

To see Cosmos DB 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 Cosmos DB, we do not need to create any schemas or secondary indices explicitly. We simply need to insert the JSON documents to a Cosmos DB 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 API 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 Cosmos DB query language through the examples we've seen so far:

- Since DocumentDB API 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…..Nodem`, 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.
- Cosmos DB 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 Cosmos DB 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.

Cosmos DB indexing

Before we get into the DocumentDB API SQL syntax, it is worth exploring the indexing design in Cosmos DB API API.

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 Cosmos DB 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 Cosmos DB 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 Cosmos DB allows the developer to make cost based tradeoffs between index overhead in relation to the query performance.

Refer to the Azure Cosmos DB samples on MSDN for samples showing how to configure the indexing policy for a collection. Let’s now get into the details of the Azure Cosmos DB SQL syntax.

Basics of an Azure Cosmos DB 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 <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 API 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 API SQL between any two JSON types.

Op	**Undefined**	**Null**	**Boolean**	**Number**	**String**	**Object**	**Array**
Undefined	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined
Null	Undefined	**OK**	Undefined	Undefined	Undefined	Undefined	Undefined
Boolean	Undefined	Undefined	**OK**	Undefined	Undefined	Undefined	Undefined

Number	Undefined	Undefined	Undefined	**OK**	Undefined	Undefined	Undefined
String	Undefined	Undefined	Undefined	Undefined	**OK**	Undefined	Undefined
Object	Undefined	Undefined	Undefined	Undefined	Undefined	**OK**	Undefined
Array	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined	**OK**

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 API 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 in others ("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 API 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.

Aggregate Functions

You can also perform aggregations in the `SELECT` clause. Aggregate functions perform a calculation on a set of values and return a single value. For example, the following query returns the count of family documents within the collection.

Query

```
SELECT COUNT(1)
FROM Families f
```

Results

```
[{
   "$1": 2
}]
```

You can also return the scalar value of the aggregate by using the `VALUE` keyword. For example, the following query returns the count of values as a single number:

Query

```
SELECT VALUE COUNT(1)
FROM Families f
```

Results

```
[ 2 ]
```

You can also perform aggregates in combination with filters. For example, the following query returns the count of documents with the address in the state of Washington.

Query

```
SELECT VALUE COUNT(1)
FROM Families f
WHERE f.address.state = "WA"
```

Results

```
[{
   "$1": 1
}]
```

The following tables shows the list of supported aggregate functions in DocumentDB API. `SUM` and `AVG` are performed over numeric values, whereas `COUNT`, `MIN`, and `MAX` can be performed over numbers, strings, Booleans, and nulls.

USAGE	DESCRIPTION
COUNT	Returns the number of items in the expression.
SUM	Returns the sum of all the values in the expression.
MIN	Returns the minimum value in the expression.
MAX	Returns the maximum value in the expression.
AVG	Returns the average of the values in the expression.

Aggregates can also be performed over the results of an array iteration. For more details, see Array Iteration in Queries.

> NOTE
>
> When using the Azure Portal's Query Explorer, note that aggregation queries may return the partially aggregated results over a query page. The SDKs will produce a single cumulative value across all pages.
>
> In order to perform aggregation queries using code, you need .NET SDK 1.12.0, .NET Core SDK 1.1.0, or Java SDK 1.9.5 or above.

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 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 API 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"
}]
```

You can also perform aggregation over the result of array iteration. For example, the following query counts the number of children among all families.

Query

```
SELECT COUNT(child)
FROM child IN Families.children
```

Results

```
[
  {
    "$1": 3
  }
]
```

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 API 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 (111) 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 112 = 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

Azure Cosmos DB 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 Azure Cosmos DB 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 API 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 DoucmentDB API SQL syntax is extended to support custom application logic using these User Defined Functions. UDFs can be registered with DocumentDB API 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 Cosmos DB 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, Cosmos DB 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 API SQL to provide a rich programmable interface to do complex procedural, conditional logic with the help of inbuilt JavaScript runtime capabilities.

DocumentDB API 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

Cosmos DB, by the virtue of being a JSON database, draws parallels with JavaScript operators and its evaluation semantics. While Cosmos DB tries to preserve JavaScript semantics in terms of JSON support, the operation evaluation deviates in some instances.

In DocumentDB API 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 API 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 API SQL.

Parameterized SQL queries

Cosmos DB 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 Cosmos DB 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 Cosmos DB is schema-less, parameters are not validated against any type.

Built-in functions

Cosmos DB 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

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
[ABS (num_expr)	Returns the absolute (positive) value of the specified numeric expression.
CEILING (num_expr)	Returns the smallest integer value greater than, or equal to, the specified numeric expression.
FLOOR (num_expr)	Returns the largest integer less than or equal to the specified numeric expression.
EXP (num_expr)	Returns the exponent of the specified numeric expression.
LOG (num_expr [,base])	Returns the natural logarithm of the specified numeric expression, or the logarithm using the specified base
LOG10 (num_expr)	Returns the base-10 logarithmic value of the specified numeric expression.
ROUND (num_expr)	Returns a numeric value, rounded to the closest integer value.
TRUNC (num_expr)	Returns a numeric value, truncated to the closest integer value.
SQRT (num_expr)	Returns the square root of the specified numeric expression.
SQUARE (num_expr)	Returns the square of the specified numeric expression.

USAGE	DESCRIPTION
POWER (num_expr, num_expr)	Returns the power of the specified numeric expression to the value specifed.
SIGN (num_expr)	Returns the sign value (-1, 0, 1) of the specified numeric expression.
ACOS (num_expr)	Returns the angle, in radians, whose cosine is the specified numeric expression; also called arccosine.
ASIN (num_expr)	Returns the angle, in radians, whose sine is the specified numeric expression. This is also called arcsine.
ATAN (num_expr)	Returns the angle, in radians, whose tangent is the specified numeric expression. This is also called arctangent.
ATN2 (num_expr)	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.
COS (num_expr)	Returns the trigonometric cosine of the specified angle, in radians, in the specified expression.
COT (num_expr)	Returns the trigonometric cotangent of the specified angle, in radians, in the specified numeric expression.
DEGREES (num_expr)	Returns the corresponding angle in degrees for an angle specified in radians.
PI ()	Returns the constant value of PI.
RADIANS (num_expr)	Returns radians when a numeric expression, in degrees, is entered.
SIN (num_expr)	Returns the trigonometric sine of the specified angle, in radians, in the specified expression.
TAN (num_expr)	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 Cosmos DB’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
IS_ARRAY (expr)	Returns a Boolean indicating if the type of the value is an array.
IS_BOOL (expr)	Returns a Boolean indicating if the type of the value is a Boolean.
IS_NULL (expr)	Returns a Boolean indicating if the type of the value is null.
IS_NUMBER (expr)	Returns a Boolean indicating if the type of the value is a number.
IS_OBJECT (expr)	Returns a Boolean indicating if the type of the value is a JSON object.
IS_STRING (expr)	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.

USAGE	DESCRIPTION
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.
LOWER (str_expr)	Returns a string expression after converting uppercase character data to lowercase.
UPPER (str_expr)	Returns a string expression after converting lowercase character data to uppercase.
REPLACE (str_expr, str_expr, str_expr)	Replaces all occurrences of a specified string value with another string value.
REPLICATE (str_expr, num_expr)	Repeats a string value a specified number of times.
REVERSE (str_expr)	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

Cosmos DB 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

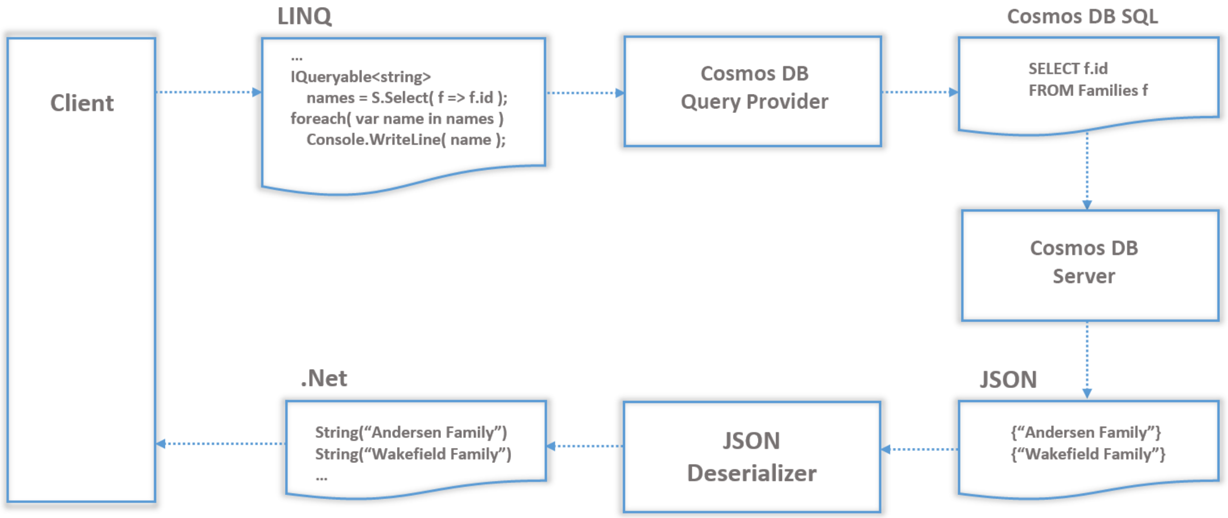
```
[{
  "id": "WakefieldFamily"
}]
```

For more details on geospatial support in Cosmos DB, please see Working with geospatial data in Azure Cosmos DB. That wraps up spatial functions, and the SQL syntax for Cosmos DB. 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 API SQL

LINQ is a .NET programming model that expresses computation as queries on streams of objects. Cosmos DB 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 Cosmos DB queries.

The picture below shows the architecture of supporting LINQ queries using Cosmos DB. Using the Cosmos DB client, developers can create an **IQueryable** object that directly queries the Cosmos DB query provider, which then translates the LINQ query into a Cosmos DB query. The query is then passed to the Cosmos DB 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 Cosmos DB query provider performs a best effort mapping from a LINQ query into a Cosmos DB 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
- **Count**, **Sum**, **Min**, **Max**, and **Average** operators for aggregation, and their async equivalents **CountAsync**, **SumAsync**, **MinAsync**, **MaxAsync**, and **AverageAsync**.
- **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 Cosmos DB 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 Cosmos DB 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

Cosmos DB exposes resources through a REST API that can be called by any language capable of making HTTP/HTTPS requests. Additionally, Cosmos DB 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 Cosmos DB database account.

REST API

Cosmos DB offers an open RESTful programming model over HTTP. Database accounts can be provisioned using an Azure subscription. The Cosmos DB 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 Cosmos DB query. Queries are always read only operations with no side-effects.

The following examples show a POST for a DocumentDB API 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. If the specified query has an aggregation function like `COUNT`, then the query page may return a partially aggregated value over the page of results. The clients must perform a second level aggregation over these results to produce the final results, for example, sum over the counts returned in the individual pages to return the total count.

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 Azure Cosmos DB Consistency Levels.

If the configured indexing policy on the collection cannot support the specified query, the Azure Cosmos DB 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 Cosmos DB can automatically extract this from the query text), and `EnableCrossPartitionQuery` to run queries that may need to be run against multiple partitions.

Refer to Azure Cosmos DB .NET samples for more samples containing queries.

NOTE

In order to perform aggregation queries, you need SDKs 1.12.0 or above. LINQ support for aggregation functions is not supported but will be available in .NET SDK 1.13.0.

JavaScript server-side API

Cosmos DB 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]);
                    }
                })
        });
}
```

References

1. Introduction to Azure Cosmos DB
2. Azure Cosmos DB SQL specification
3. Azure Cosmos DB .NET samples
4. Azure Cosmos DB 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.

Partitioning in Azure Cosmos DB using the DocumentDB API

5/30/2017 • 7 min to read • Edit Online

Microsoft Azure Cosmos DB is a global distributed, multi-model database service 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 to work with partitioning of Cosmos DB containers with the DocumentDB API. See partitioning and horizontal scaling for an overview of concepts and best practices for partitioning with any Azure Cosmos DB API.

To get started with code, download the project from Github.

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

- How does partitioning work in Azure Cosmos DB?
- How do I configure partitioning in Azure Cosmos DB
- What are partition keys, and how do I pick the right partition key for my application?

To get started with code, download the project from Azure Cosmos DB Performance Testing Driver Sample.

Partition keys

In the DocumentDB API, you specify the partition key definition in the form of a JSON path. The following table shows examples of partition key definitions and the values corresponding to each. The partition key is specified as a path, e.g. `/department` represents the property department.

Partition Key	**Description**
/department	Corresponds to the value of doc.department where doc is the item.
/properties/name	Corresponds to the value of doc.properties.name where doc is the item (nested property).
/id	Corresponds to the value of doc.id (id and partition key are the same property).
/"department name"	Corresponds to the value of doc["department name"] where doc is the item.

NOTE

The syntax for partition key 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 is implicitly indexed and cannot be excluded from indexing using indexing policy overrides.

Let's look at how the choice of partition key impacts the performance of your application.

Working with the DocumentDB SDKs

Azure Cosmos DB added support for automatic partitioning with REST API version 2015-12-16. In order to create partitioned containers, you must download SDK versions 1.6.0 or newer in one of the supported SDK platforms (.NET, Node.js, Java, Python, MongoDB).

Creating containers

The following sample shows a .NET snippet to create a container 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 container 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" });

// Container for device telemetry. Here the 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 });
```

This method makes a REST API call to Cosmos DB, and the service will provision a number of partitions based on the requested throughput. You can change the throughput of a container as your performance needs evolve.

Reading and writing items

Now, let's insert data into Cosmos DB. Here's a sample class containing a device reading, and a call to CreateDocumentAsync to insert a new device reading into a container. This is an example leveraging the DocumentDB API:

```
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 item 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 containers

When you query data in partitioned containers, Cosmos DB 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);
```

Cosmos DB supports aggregate functions `COUNT`, `MIN`, `MAX`, `SUM` and `AVG` over partitioned containers using SQL starting with SDKs 1.12.0 and above. Queries must include a single aggregate operator, and must include a single value in the projection.

Parallel query execution

The Cosmos DB 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 container'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 container'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 item.

```
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 containers from single-partition containers.

Next steps

In this article, we provided an overview of how to work with partitioning of Cosmos DB containers with the DocumentDB API. Also see partitioning and horizontal scaling for an overview of concepts and best practices for partitioning with any Azure Cosmos DB API.

- Perform scale and performance testing with Cosmos DB. See Performance and Scale Testing with Azure Cosmos DB for a sample.
- Get started coding with the SDKs or the REST API
- Learn about provisioned throughput in Azure Cosmos DB

Azure Cosmos DB server-side programming: Stored procedures, database triggers, and UDFs

6/13/2017 • 25 min to read • Edit Online

Learn how Azure Cosmos DB’s language integrated, transactional execution of JavaScript lets developers write **stored procedures**, **triggers** and **user defined functions (UDFs)** natively in an ECMAScript 2015 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 Cosmos DB'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 a stored procedure, trigger, or UDF using JavaScript?
- How does Cosmos DB guarantee ACID?
- How do transactions work in Cosmos DB?
- 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 Cosmos DB 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:** Cosmos DB 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 Cosmos DB 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 – Cosmos DB 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, Azure Cosmos DB 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",
    serverScript: 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 Cosmos DB 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 Azure Cosmos DB 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 Cosmos DB resources.

```
var createDocumentStoredProc = {
    id: "createMyDocument",
    serverScript: 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 Azure Cosmos DB 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 Cosmos DB (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 Cosmos DB, 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 Cosmos DB 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 = {
    id: "exchangeItems",
    serverScript: 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 Azure Cosmos DB Partitioning.

Commit and rollback

Transactions are deeply and natively integrated into Cosmos DB'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 Cosmos DB.

If there is any exception that's propagated from the script, Cosmos DB'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 Cosmos DB.

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 Cosmos DB 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. Cosmos DB 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

Cosmos DB 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 = {
    id: "validateDocumentContents",
    serverScript: 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 = {
    id: "updateMetadata",
    serverScript: 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 Cosmos DB. 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 API 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 Cosmos DB 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 = {
   id: "tax",
   serverScript: 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 JSDocs.

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 funciton 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

SQL	JAVASCRIPT QUERY API	DESCRIPTION BELOW

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 API 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 dbs/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: ‘dbs/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 Data 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
- Secure and Portable Database Extensibility
- Service Oriented Database Architecture
- Hosting the .NET Runtime in Microsoft SQL server

Performance and scale testing with Azure Cosmos DB

6/6/2017 • 4 min to read • Edit Online

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 Cosmos DB 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 Cosmos DB workloads, or evaluating Cosmos DB 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 Cosmos DB?
- How do I achieve high throughput levels with Cosmos DB from my client application?

To get started with code, please download the project from Azure Cosmos DB Performance Testing Sample.

> NOTE
>
> The goal of this application is to demonstrate best practices for extracting better performance out of Cosmos DB 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 Cosmos DB performance, see Azure Cosmos DB performance tips.

Run the performance testing application

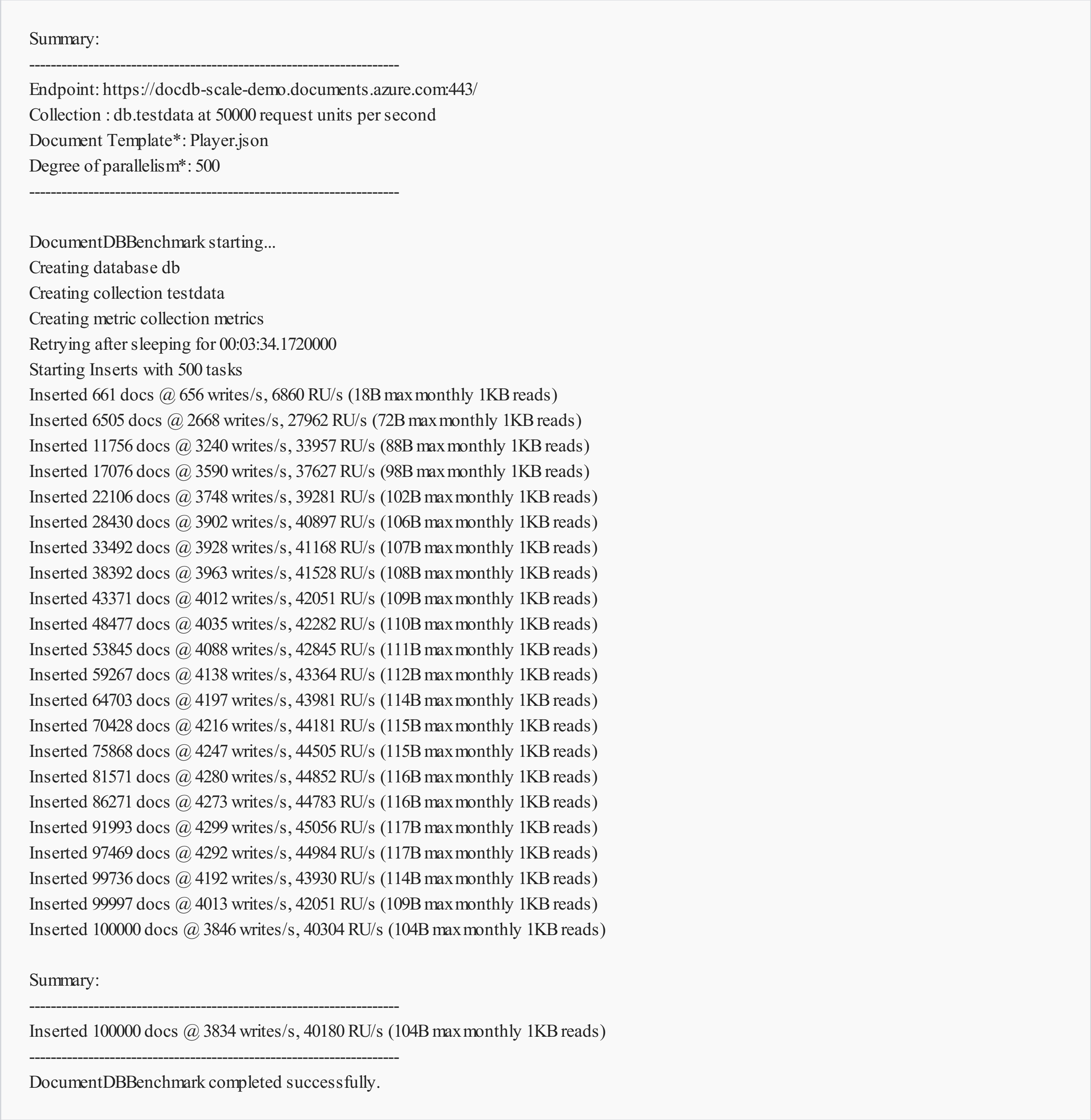
The quickest way to get started is to compile and run the .NET sample below, as described in the steps below. You can also review the source code and implement similar configurations to your own client applications.

Step 1: Download the project from Azure Cosmos DB 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. Cosmos DB 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 00:03:34.1720000
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 askcosmosdb@microsoft.com or file a support ticket from the Azure Portal.

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 Cosmos DB using a .NET console app. Please refer to the links below for additional information on working with Cosmos DB.

- Azure Cosmos DB performance testing sample
- Client configuration options to improve Azure Cosmos DB performance
- Server-side partitioning in Azure Cosmos DB

- DocumentDB collections and performance levels
- DocumentDB .NET SDK documentation on MSDN
- DocumentDB .NET samples
- Azure Cosmos DB blog on performance tips

Performance tips for Azure Cosmos DB

5/30/2017 • 14 min to read • Edit Online

Azure Cosmos DB 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 Cosmos DB. Scaling up and down is as easy as making a single API call or SDK method call. However, because Cosmos DB 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 Cosmos DB 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 Cosmos DB. Because of this, Direct Mode offers better performance due to fewer network hops.

2. **Connection policy: Use the TCP protocol**

   When using Direct Mode, there are two protocol options available:

   - TCP
   - HTTPS

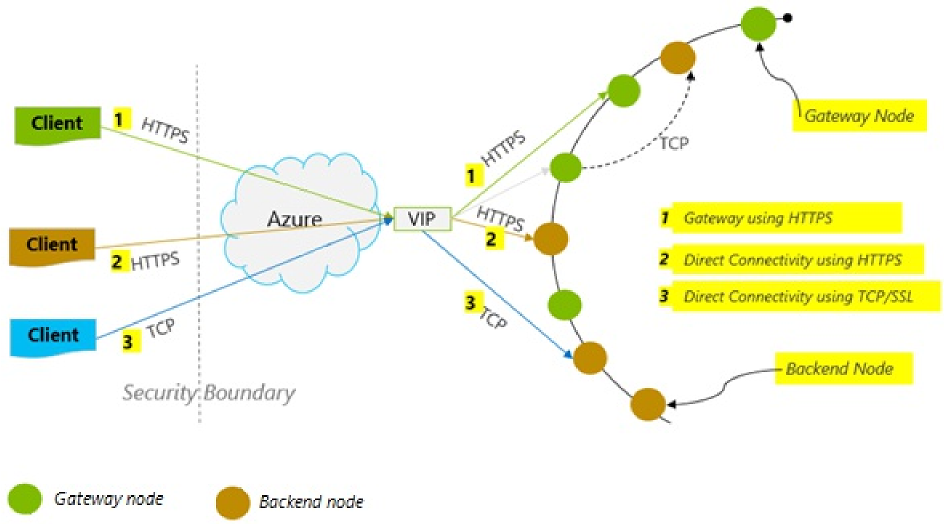
     Cosmos DB offers a simple and open RESTful programming model over HTTPS. Additionally, it offers an efficient TCP protocol, which is also RESTful in its communication model and is available through the .NET client SDK. Both Direct TCP and HTTPS use SSL for initial authentication and encrypting traffic. For best performance, use the TCP protocol when possible.

     When using TCP in Gateway Mode, TCP Port 443 is the Cosmos DB port, and 10250 is the MongoDB API port. When using TCP in Direct Mode, in addition to the Gateway ports, you need to ensure the port range between 10000 and 20000 is open because Cosmos DB uses dynamic TCP ports. If these ports are not open and you attempt to use TCP, you 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 the Azure 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.



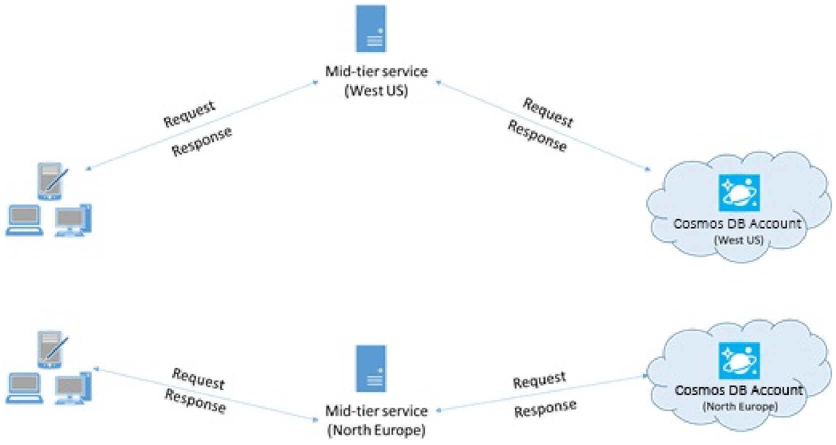
3. **Call OpenAsync to avoid startup latency on first request**

   By default, the first request has 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 Cosmos DB in the same region as the Cosmos DB database. For an approximate comparison, calls to Cosmos DB 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 Cosmos DB endpoint. For a list of available regions, see Azure Regions.



5. **Increase number of threads/tasks**

   Since calls to Azure Cosmos DB 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 Cosmos DB.

SDK Usage

1. **Install the most recent SDK**

   The Cosmos DB SDKs are constantly being improved to provide the best performance. See the Cosmos DB SDK pages to determine the most recent SDK and review improvements.

2. **Use a singleton Cosmos DB 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 when using Gateway mode**

   Cosmos DB requests are made over HTTPS/REST when using Gateway mode, 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 Cosmos DB. 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 then 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 chooses 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 number 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 Cosmos DB 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 (for example, 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, for example, 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 Cosmos DB 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.

11. **Use 64-bit host processing**

    The DocumentDB SDK works in a 32-bit host process when you are using DocumentDB .NET SDK version 1.11.4 and above. However, if you are using cross partition queries, 64-bit host processing is recommended for improved performance. The following types of applications have 32-bit host process as the default, so in order to change that to 64-bit, follow these steps based on the type of your application:

    - For Executable applications, this can be done by unchecking the **Prefer 32-bit** option in the **Project Properties** window, on the **Build** tab.

    - For VSTest based test projects, this can be done by selecting **Test**->**Test Settings**->**Default Processor Architecture as X64**, from the **Visual Studio Test** menu option.

    - For locally deployed ASP.NET Web applications, this can be done by checking the **Use the 64-bit version of IIS Express for web sites and projects**, under **Tools**->**Options**->**Projects and Solutions**->**Web Projects**.

    - For ASP.NET Web applications deployed on Azure, this can be done by choosing the **Platform as 64-bit** in the **Application Settings** on the Azure portal.

Indexing Policy

1. **Use lazy indexing for faster peak time ingestion rates**

   Cosmos DB 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 are eventually consistent regardless of the consistency level configured for the Cosmos DB 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**

   Cosmos DB’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 Azure Cosmos DB indexing policies.

Throughput

1. **Measure and tune for lower request units/second usage**

   Cosmos DB 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 varies 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 preceding query returns 1000 1KB-documents, the cost of the operation is 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 is 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 Cosmos DB for high-performance scenarios on a few client machines, see Performance and scale testing with Cosmos DB.

Also, to learn more about designing your application for scale and high performance, see Partitioning and scaling in Azure Cosmos DB.

Multi-master globally replicated database architectures with Azure Cosmos DB

5/30/2017 • 7 min to read • Edit Online

Azure Cosmos DB supports turnkey global replication, which allows you to distribute data to multiple regions with low latency access anywhere in the workload. This model is commonly used for publisher/consumer workloads where there is a writer in a single geographic region and globally distributed readers in other (read) regions.

You can also use Azure Cosmos DB's global replication support to build applications in which writers and readers are globally distributed. This document outlines a pattern that enables achieving local write and local read access for distributed writers using Azure Cosmos DB.

Content Publishing - an example scenario

Let's look at a real world scenario to describe how you can use globally distributed multi-region/multi-master read write patterns with Azure Cosmos DB. Consider a content publishing platform built on Azure Cosmos DB. Here are some requirements that this platform must meet for a great user experience for both publishers and consumers.

- Both authors and subscribers are spread over the world
- Authors must publish (write) articles to their local (closest) region
- Authors have readers/subscribers of their articles who are distributed across the globe.
- Subscribers should get a notification when new articles are published.
- Subscribers must be able to read articles from their local region. They should also be able to add reviews to these articles.
- Anyone including the author of the articles should be able view all the reviews attached to articles from a local region.

Assuming millions of consumers and publishers with billions of articles, soon we have to confront the problems of scale along with guaranteeing locality of access. As with most scalability problems, the solution lies in a good partitioning strategy. Next, let's look at how to model articles, review, and notifications as documents, configure Azure Cosmos DB accounts, and implement a data access layer.

If you would like to learn more about partitioning and partition keys, see Partitioning and Scaling in Azure Cosmos DB.

Modeling notifications

Notifications are data feeds specific to a user. Therefore, the access patterns for notifications documents are always in the context of single user. For example, you would "post a notification to a user" or "fetch all notifications for a given user". So, the optimal choice of partitioning key for this type would be `UserId`.

```
class Notification
{
    // Unique ID for Notification.
    public string Id { get; set; }

    // The user Id for which notification is addressed to.
    public string UserId { get; set; }

    // The partition Key for the resource.
    public string PartitionKey
    {
        get
        {
            return this.UserId;
        }
    }

    // Subscription for which this notification is raised.
    public string SubscriptionFilter { get; set; }

    // Subject of the notification.
    public string ArticleId { get; set; }
}
```

Modeling subscriptions

Subscriptions can be created for various criteria like a specific category of articles of interest, or a specific publisher. Hence the `SubscriptionFilter` is a good choice for partition key.

```
class Subscriptions
{
    // Unique ID for Subscription
    public string Id { get; set; }

    // Subscription source. Could be Author | Category etc.
    public string SubscriptionFilter { get; set; }

    // subscribing User.
    public string UserId { get; set; }

    public string PartitionKey
    {
        get
        {
            return this.SubscriptionFilter;
        }
    }
}
```

Modeling articles

Once an article is identified through notifications, subsequent queries are typically based on the `Article.Id`. Choosing `Article.Id` as partition the key thus provides the best distribution for storing articles inside an Azure Cosmos DB collection.

```
class Article
{
    // Unique ID for Article
    public string Id { get; set; }

    public string PartitionKey
    {
        get
        {
            return this.Id;
        }
    }

    // Author of the article
    public string Author { get; set; }

    // Category/genre of the article
    public string Category { get; set; }

    // Tags associated with the article
    public string[] Tags { get; set; }

    // Title of the article
    public string Title { get; set; }

    //...
}
```

Modeling reviews

Like articles, reviews are mostly written and read in the context of article. Choosing `ArticleId` as a partition key provides best distribution and efficient access of reviews associated with article.

```
class Review
{
    // Unique ID for Review
    public string Id { get; set; }

    // Article Id of the review
    public string ArticleId { get; set; }

    public string PartitionKey
    {
        get
        {
            return this.ArticleId;
        }
    }

    //Reviewer Id
    public string UserId { get; set; }
    public string ReviewText { get; set; }

    public int Rating { get; set; } }
}
```

Data access layer methods

Now let's look at the main data access methods we need to implement. Here's the list of methods that the `ContentPublishDatabase` needs:

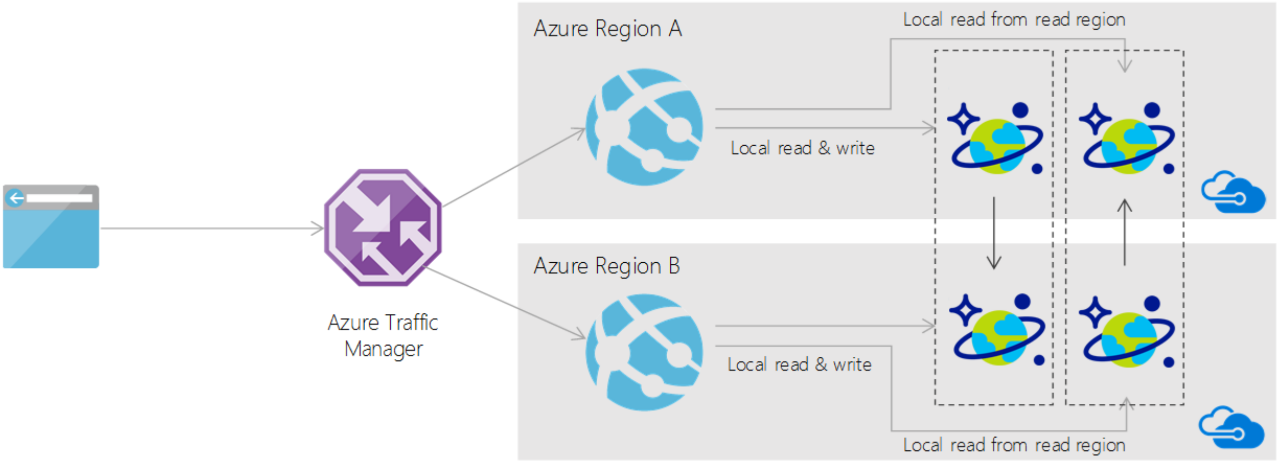
```
class ContentPublishDatabase
{
    public async Task CreateSubscriptionAsync(string userId, string category);

    public async Task<IEnumerable<Notification>> ReadNotificationFeedAsync(string userId);

    public async Task<Article> ReadArticleAsync(string articleId);

    public async Task WriteReviewAsync(string articleId, string userId, string reviewText, int rating);

    public async Task<IEnumerable<Review>> ReadReviewsAsync(string articleId);
}
```

Azure Cosmos DB account configuration

To guarantee local reads and writes, we must partition data not just on partition key, but also based on the geographical access pattern into regions. The model relies on having a geo-replicated Azure Cosmos DB database account for each region. For example, with two regions, here's a setup for multi-region writes:

ACCOUNT NAME	WRITE REGION	READ REGION
`contentpubdatabase-usa.documents.azure.com`	`West US`	`North Europe`
`contentpubdatabase-europe.documents.azure.com`	`North Europe`	`West US`

The following diagram shows how reads and writes are performed in a typical application with this setup:



Here is a code snippet showing how to initialize the clients in a DAL running in the `West US` region.

```
ConnectionPolicy writeClientPolicy = new ConnectionPolicy { ConnectionMode = ConnectionMode.Direct, ConnectionProtocol = Protocol.Tcp };
writeClientPolicy.PreferredLocations.Add(LocationNames.WestUS);
writeClientPolicy.PreferredLocations.Add(LocationNames.NorthEurope);

DocumentClient writeClient = new DocumentClient(
    new Uri("https://contentpubdatabase-usa.documents.azure.com"),
    writeRegionAuthKey,
    writeClientPolicy);

ConnectionPolicy readClientPolicy = new ConnectionPolicy { ConnectionMode = ConnectionMode.Direct, ConnectionProtocol = Protocol.Tcp };
readClientPolicy.PreferredLocations.Add(LocationNames.NorthEurope);
readClientPolicy.PreferredLocations.Add(LocationNames.WestUS);

DocumentClient readClient = new DocumentClient(
    new Uri("https://contentpubdatabase-europe.documents.azure.com"),
    readRegionAuthKey,
    readClientPolicy);
```

With the preceding setup, the data access layer can forward all writes to the local account based on where it is deployed. Reads are performed by reading from both accounts to get the global view of data. This approach can be extended to as many regions as required. For example, here's a setup with three geographic regions:

ACCOUNT NAME	WRITE REGION	READ REGION 1	READ REGION 2
`contentpubdatabase-usa.documents.azure.com`	`West US`	`North Europe`	`Southeast Asia`
`contentpubdatabase-europe.documents.azure.com`	`North Europe`	`West US`	`Southeast Asia`
`contentpubdatabase-asia.documents.azure.com`	`Southeast Asia`	`North Europe`	`West US`

Data access layer implementation

Now let's look at the implementation of the data access layer (DAL) for an application with two writable regions. The DAL must implement the following steps:

- Create multiple instances of `DocumentClient` for each account. With two regions, each DAL instance has one `writeClient` and one `readClient`.
- Based on the deployed region of the application, configure the endpoints for `writeclient` and `readClient`. For example, the DAL deployed in `West US` uses `contentpubdatabase-usa.documents.azure.com` for performing writes. The DAL deployed in `NorthEurope` uses `contentpubdatabase-europ.documents.azure.com` for writes.

With the preceding setup, the data access methods can be implemented. Write operations forward the write to the corresponding `writeClient`.

```
public async Task CreateSubscriptionAsync(string userId, string category)
{
    await this.writeClient.CreateDocumentAsync(this.contentCollection, new Subscriptions
    {
        UserId = userId,
        SubscriptionFilter = category
    });
}

public async Task WriteReviewAsync(string articleId, string userId, string reviewText, int rating)
{
    await this.writeClient.CreateDocumentAsync(this.contentCollection, new Review
    {
        UserId = userId,
        ArticleId = articleId,
        ReviewText = reviewText,
        Rating = rating
    });
}
```

For reading notifications and reviews, you must read from both regions and union the results as shown in the following snippet:

```
public async Task<IEnumerable<Notification>> ReadNotificationFeedAsync(string userId)
{
    IDocumentQuery<Notification> writeAccountNotification = (
        from notification in this.writeClient.CreateDocumentQuery<Notification>(this.contentCollection)
        where notification.UserId == userId
        select notification).AsDocumentQuery();

    IDocumentQuery<Notification> readAccountNotification = (
        from notification in this.readClient.CreateDocumentQuery<Notification>(this.contentCollection)
        where notification.UserId == userId
        select notification).AsDocumentQuery();

    List<Notification> notifications = new List<Notification>();

    while (writeAccountNotification.HasMoreResults || readAccountNotification.HasMoreResults)
    {
        IList<Task<FeedResponse<Notification>>> results = new List<Task<FeedResponse<Notification>>>();

        if (writeAccountNotification.HasMoreResults)
        {
            results.Add(writeAccountNotification.ExecuteNextAsync<Notification>());
        }

        if (readAccountNotification.HasMoreResults)
        {
            results.Add(readAccountNotification.ExecuteNextAsync<Notification>());
        }

        IList<FeedResponse<Notification>> notificationFeedResult = await Task.WhenAll(results);

        foreach (FeedResponse<Notification> feed in notificationFeedResult)
        {
            notifications.AddRange(feed);
        }
    }
    return notifications;
}

public async Task<IEnumerable<Review>> ReadReviewsAsync(string articleId)
{
    IDocumentQuery<Review> writeAccountReviews = (
        from review in this.writeClient.CreateDocumentQuery<Review>(this.contentCollection)
        where review.ArticleId == articleId
        where review.ArticleId == articleId
        select review).AsDocumentQuery();

    IDocumentQuery<Review> readAccountReviews = (
        from review in this.readClient.CreateDocumentQuery<Review>(this.contentCollection)
        where review.ArticleId == articleId
        select review).AsDocumentQuery();

    List<Review> reviews = new List<Review>();

    while (writeAccountReviews.HasMoreResults || readAccountReviews.HasMoreResults)
    {
        IList<Task<FeedResponse<Review>>> results = new List<Task<FeedResponse<Review>>>();

        if (writeAccountReviews.HasMoreResults)
        {
            results.Add(writeAccountReviews.ExecuteNextAsync<Review>());
        }

        if (readAccountReviews.HasMoreResults)
        {
            results.Add(readAccountReviews.ExecuteNextAsync<Review>());
        }

        IList<FeedResponse<Review>> notificationFeedResult = await Task.WhenAll(results);

        foreach (FeedResponse<Review> feed in notificationFeedResult)
        {
            reviews.AddRange(feed);
        }
    }

    return reviews;
}
```

Thus, by choosing a good partitioning key and static account-based partitioning, you can achieve multi-region local writes and reads using Azure Cosmos DB.

Next steps

In this article, we described how you can use globally distributed multi-region read write patterns with Azure Cosmos DB using content publishing as a sample scenario.

- Learn about how Azure Cosmos DB supports global distribution
- Learn about automatic and manual failovers in Azure Cosmos DB
- Learn about global consistency with Azure Cosmos DB
- Develop with multiple regions using the Azure Cosmos DB - DocumentDB API
- Develop with multiple regions using the Azure Cosmos DB - MongoDB API
- Develop with multiple regions using the Azure Cosmos DB - Table API

Working with Dates in Azure Cosmos DB

5/30/2017 • 2 min to read • Edit Online

Azure Cosmos DB delivers schema flexibility and rich indexing via a native JSON data model. All Azure Cosmos DB resources including databases, collections, documents, and stored procedures are modeled and stored as JSON documents. As a requirement for being portable, JSON (and Azure Cosmos DB) supports only a small set of basic types: String, Number, Boolean, Array, Object, and Null. However, JSON is flexible and allow developers and frameworks to represent more complex types using these primitives and composing them as objects or arrays.

In addition to the basic types, many applications need the DateTime type to represent dates and timestamps. This article describes how developers can store, retrieve, and query dates in Cosmos DB using the .NET SDK.

Storing DateTimes

By default, the Azure Cosmos DB SDK serializes DateTime values as ISO 8601 strings. Most applications can use the default string representation for DateTime for the following reasons:

- Strings can be compared, and the relative ordering of the DateTime values is preserved when they are transformed to strings.
- This approach doesn't require any custom code or attributes for JSON conversion.
- The dates as stored in JSON are human readable.
- This approach can take advantage of Cosmos DB's index for fast query performance.

For example, the following snippet stores an `Order` object containing two DateTime properties - `ShipDate` and `OrderDate` as a document using the .NET SDK:

```
public class Order
{
    [JsonProperty(PropertyName="id")]
    public string Id { get; set; }
    public DateTime OrderDate { get; set; }
    public DateTime ShipDate { get; set; }
    public double Total { get; set; }
}

await client.CreateDocumentAsync("/dbs/orderdb/colls/orders",
    new Order
    {
        Id = "09152014101",
        OrderDate = DateTime.UtcNow.AddDays(-30),
        ShipDate = DateTime.UtcNow.AddDays(-14),
        Total = 113.39
    });
```

This document is stored in Cosmos DB as follows:

```
{
    "id": "09152014101",
    "OrderDate": "2014-09-15T23:14:25.7251173Z",
    "ShipDate": "2014-09-30T23:14:25.7251173Z",
    "Total": 113.39
}
```

Alternatively, you can store DateTimes as Unix timestamps, that is, as a number representing the number of elapsed seconds since January 1, 1970. Cosmos DB's internal Timestamp (`_ts`) property follows this approach. You can use the UnixDateTimeConverter class to serialize DateTimes as numbers.

Indexing DateTimes for range queries

Range queries are common with DateTime values. For example, if you need to find all orders created since yesterday, or find all orders shipped in the last five minutes, you need to perform range queries. To execute these queries efficiently, you must configure your collection for Range indexing on strings.

```
DocumentCollection collection = new DocumentCollection { Id = "orders" };
collection.IndexingPolicy = new IndexingPolicy(new RangeIndex(DataType.String) { Precision = -1 });
await client.CreateDocumentCollectionAsync("/dbs/orderdb", collection);
```

You can learn more about how to configure indexing policies at Cosmos DB Indexing Policies.

Querying DateTimes in LINQ

The DocumentDB .NET SDK automatically supports querying data stored in DocumentDB via LINQ. For example, the following snippet shows a LINQ query that filters orders that were shipped in the last three days.

```
IQueryable<Order> orders = client.CreateDocumentQuery<Order>("/dbs/orderdb/colls/orders")
    .Where(o => o.ShipDate >= DateTime.UtcNow.AddDays(-3));

// Translated to the following SQL statement and executed on Cosmos DB
SELECT * FROM root WHERE (root["ShipDate"] >= "2016-12-18T21:55:03.45569Z")
```

You can learn more about Cosmos DB's SQL query language and the LINQ provider at Querying Cosmos DB.

In this article, we looked at how to store, index, and query DateTimes in Cosmos DB.

Next Steps

- Download and run the Code samples on GitHub
- Learn more about DocumentDB API Query
- Learn more about Azure Cosmos DB Indexing Policies

Modeling document data for NoSQL databases

5/30/2017 • 14 min to read • Edit Online

While schema-free databases, like Azure Cosmos DB, 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?

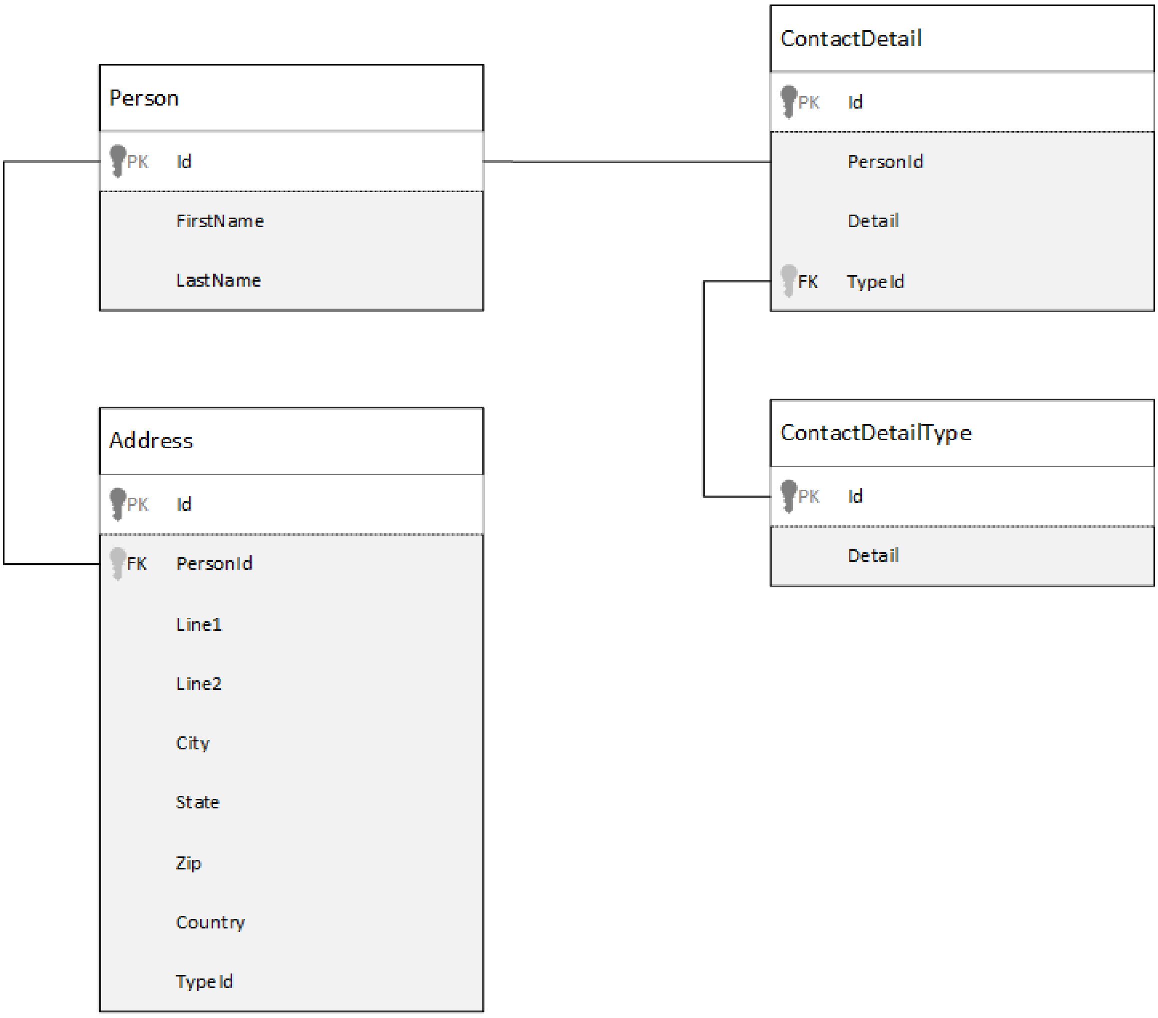
After reading this article, you will be able to answer the following questions:

- 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 Azure Cosmos DB, 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 on 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.

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 Azure Cosmos DB, 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 Azure Cosmos DB.

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": "Azure Cosmos DB 101" }
{"id": "2", "name": "Azure Cosmos DB for RDBMS Users" }
{"id": "3", "name": "Taking over the world one JSON doc at a time" }
...
{"id": "100", "name": "Learn about Azure Cosmos DB" }
...
{"id": "1000", "name": "Deep Dive in to Azure Cosmos DB" }
```

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.

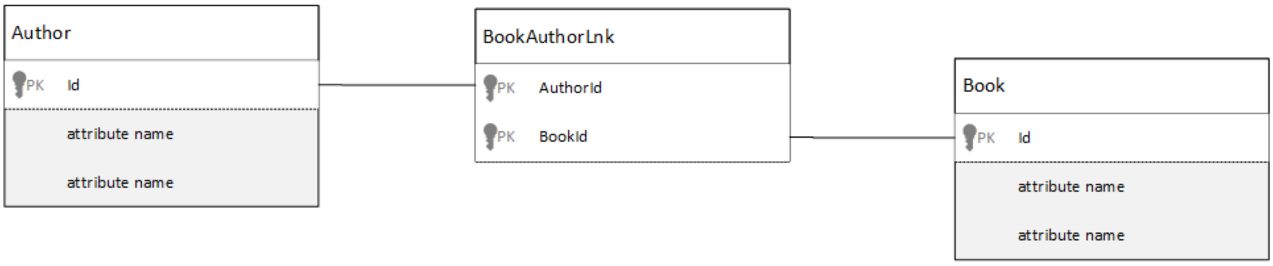
Switching things around a bit would result in a model that still represents the same data but now avoids these large mutable collections.

```
Publisher document:
{
    "id": "mspress",
    "name": "Microsoft Press"
}

Book documents:
{"id": "1","name": "Azure Cosmos DB 101", "pub-id": "mspress"}
{"id": "2","name": "Azure Cosmos DB 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 Cosmos DB", "pub-id": "mspress"}
...
{"id": "1000","name": "Deep Dive in to Azure Cosmos DB", "pub-id": "mspress"}
```

In the above example, we have dropped the unbounded collection on the publisher document. Instead we just have a 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": "Azure Cosmos DB 101" }
{"id": "b2", "name": "Azure Cosmos DB for RDBMS Users" }
{"id": "b3", "name": "Taking over the world one JSON doc at a time" }
{"id": "b4", "name": "Learn about Azure Cosmos DB" }
{"id": "b5", "name": "Deep Dive in to Azure Cosmos DB" }

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": "Azure Cosmos DB 101", "authors": ["a1", "a2"]}
{"id": "b2", "name": "Azure Cosmos DB for RDBMS Users", "authors": ["a1"]}
{"id": "b3", "name": "Learn about Azure Cosmos DB", "authors": ["a1"]}
{"id": "b4", "name": "Deep Dive in to Azure Cosmos DB", "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": "Azure Cosmos DB 101",
  "authors": [
    {"id": "a1", "name": "Thomas Andersen", "thumbnailUrl": "http://....png"},
    {"id": "a2", "name": "William Wakefield", "thumbnailUrl": "http://....png"}
  ]
},
{
  "id": "b2",
  "name": "Azure Cosmos DB 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 an 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 Azure Cosmos DB 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 Azure Cosmos DB, 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 Cosmos DB, refer to the service's documentation page.

To understand how to shard your data across multiple partitions, refer to Partitioning Data in Azure Cosmos DB.

Azure Cosmos DB: DocumentDB API getting started tutorial

6/6/2017 • 15 min to read • Edit Online

Welcome to the Azure Cosmos DB DocumentDB API getting started tutorial! After following this tutorial, you'll have a console application that creates and queries DocumentDB resources.

We'll cover:

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

Don't have time? Don't worry! The complete solution is available on GitHub. Jump to the Get the complete NoSQL tutorial 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

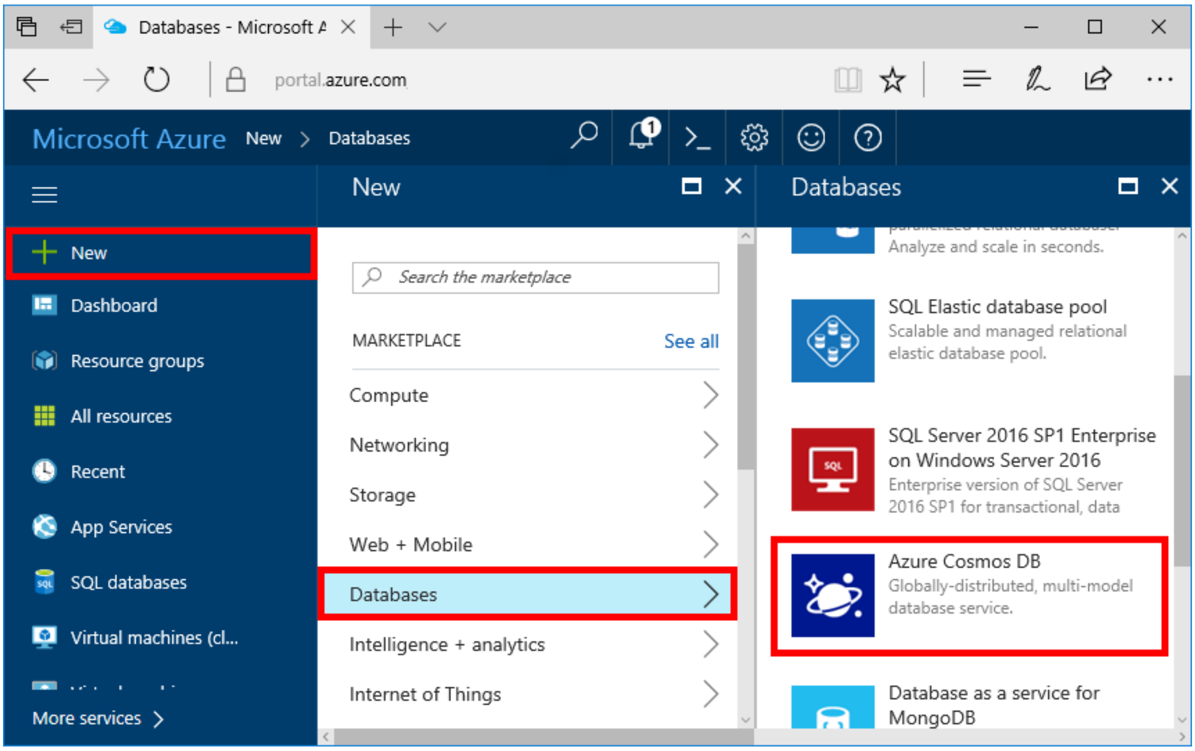
Please make sure you have the following:

- An active Azure account. If you don't have one, you can sign up for a free account.
  - Alternatively, you can use the Azure Cosmos DB Emulator for this tutorial.
- Visual Studio 2013 / Visual Studio 2015.

Step 1: Create an Azure Cosmos DB account

Let's create an Azure Cosmos DB 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 Azure Cosmos DB Emulator, please follow the steps at Azure Cosmos DB Emulator to setup the emulator and skip ahead to Setup your Visual Studio Solution.

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

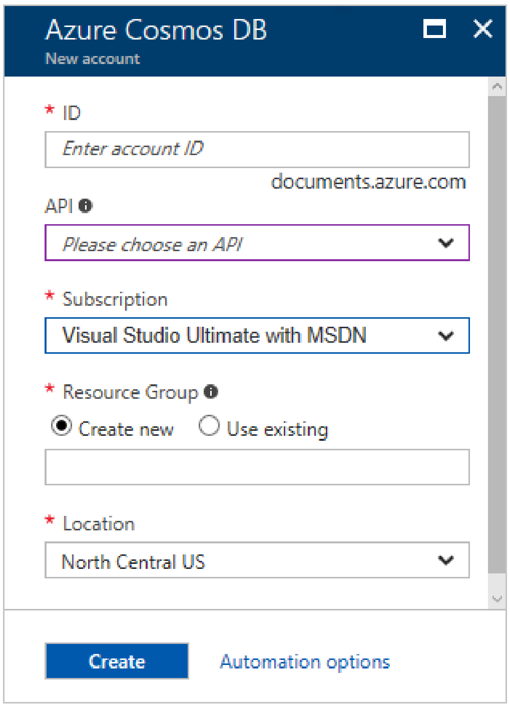


3. On the **New account** blade, specify the configuration that you want for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article we program against the DocumentDB API, so choose **SQL (DocumentDB)** as you fill out the form. But if you have graph data for a social media app, or key/value (table) data, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

   Complete the fields on the **New account** blade, using the information in the following screenshot as a guide. When you set up your account, choose unique values that do not match those in the screenshot.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

* Subscription
Visual Studio Ultimate with MSDN

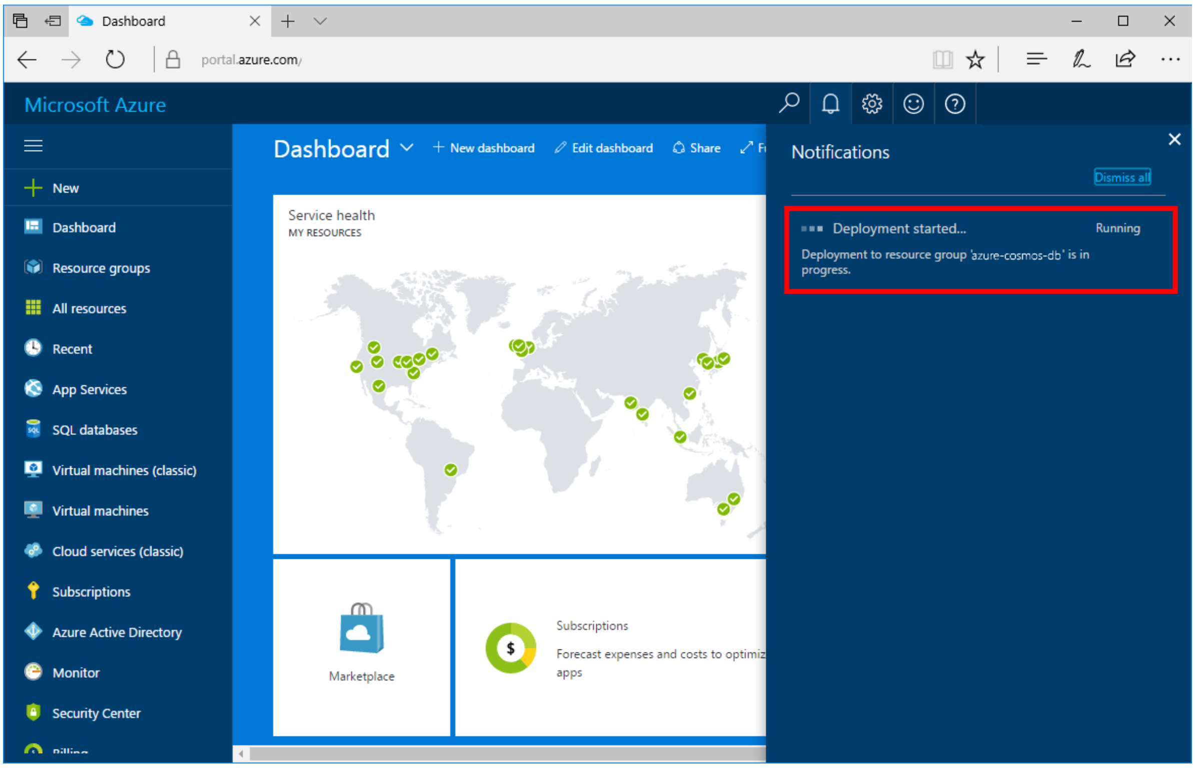
* Resource Group
● Create new ○ Use existing

* Location
North Central US

Create Automation options

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that identifies your Azure Cosmos DB account. The string *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 through 50 characters.
API	SQL (DocumentDB)	We program against the DocumentDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for your Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource-group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location that's closest to your users to give them the fastest access to the data.

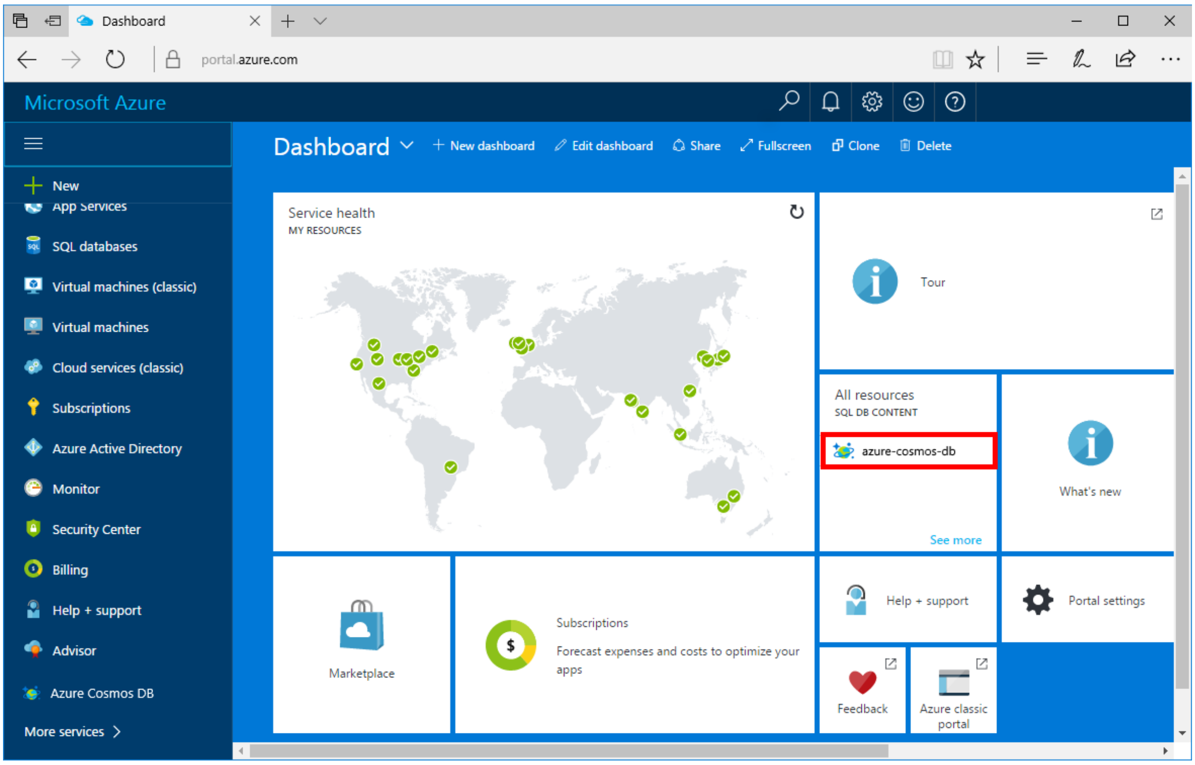
4. Click **Create** to create the account.
5. On the top toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

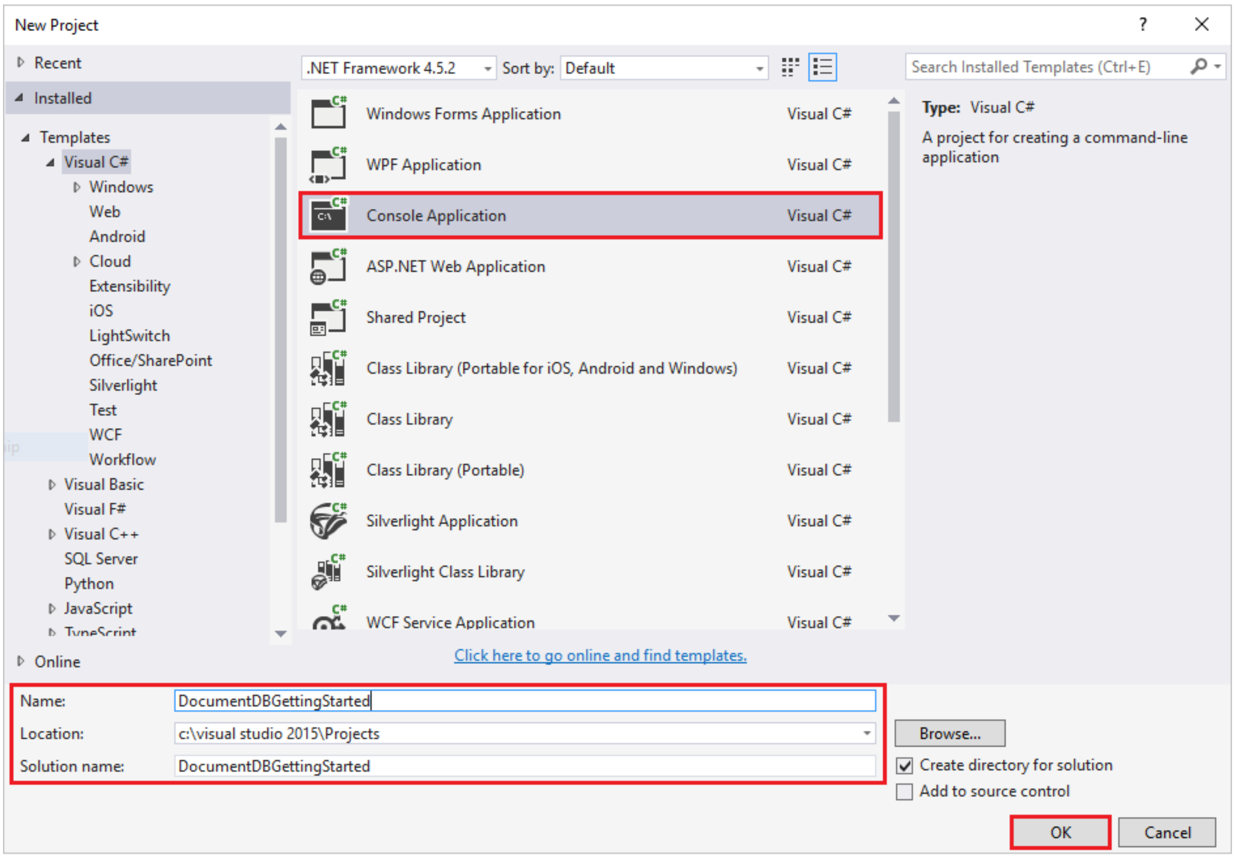
Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the **All Resources** tile.



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**.


New Project
Recent
Installed
Templates
Visual C#
Windows
Web
Android
Cloud
Extensibility
iOS
LightSwitch
Office/SharePoint
Silverlight
Test
WCF
Workflow
Visual Basic
Visual F#
Visual C++
SQL Server
Python
JavaScript
TypeScript
Online
.NET Framework 4.5.2
Sort by: Default
Search Installed Templates (Ctrl+E)
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
ASP.NET Web Application Visual C#
Shared Project Visual C#
Class Library (Portable for iOS, Android and Windows) Visual C#
Class Library Visual C#
Class Library (Portable) Visual C#
Silverlight Application Visual C#
Silverlight Class Library Visual C#
WCF Service Application Visual C#
Click here to go online and find templates.
Browse...
OK
Cancel

Type: Visual C#

A project for creating a command-line application

Name: DocumentDBGettingStarted

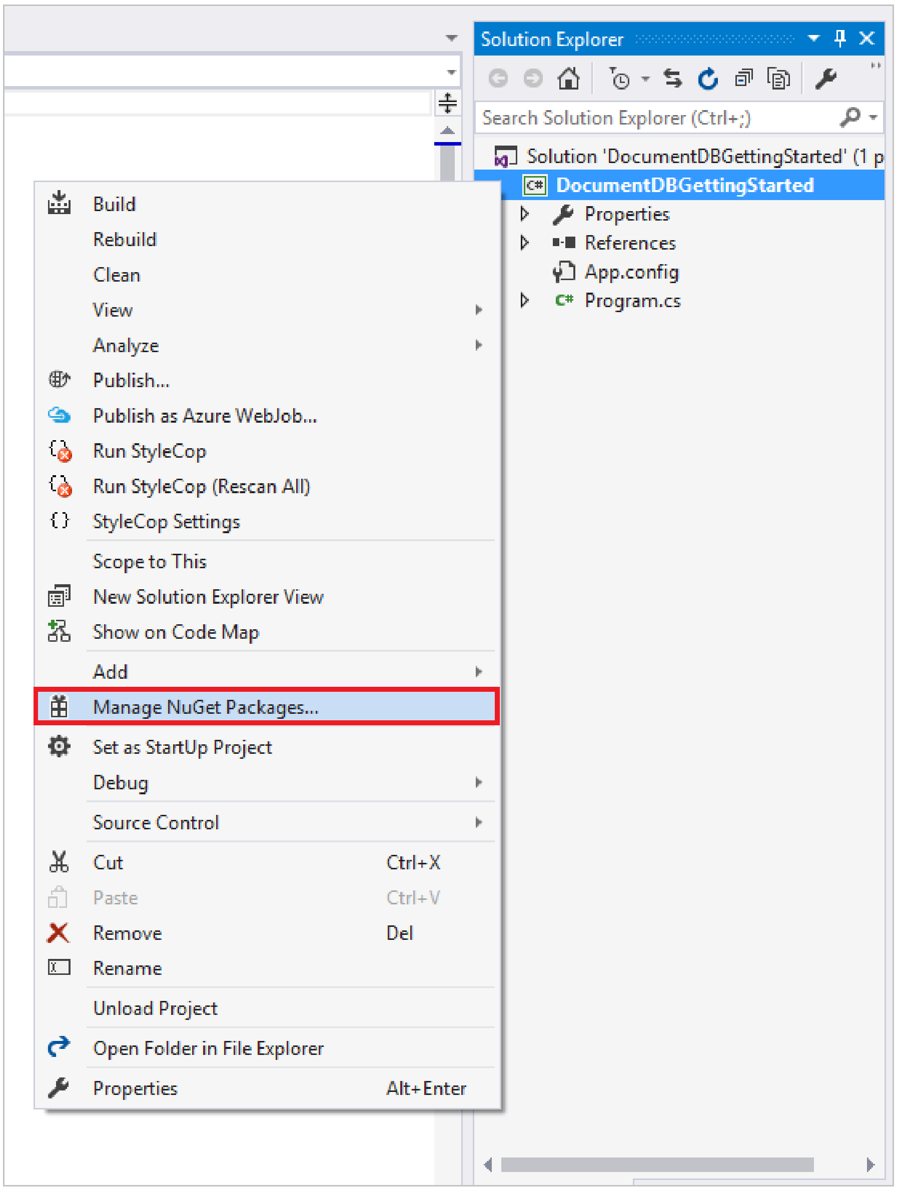
Location: c:\visual studio 2015\Projects

Solution name: DocumentDBGettingStarted

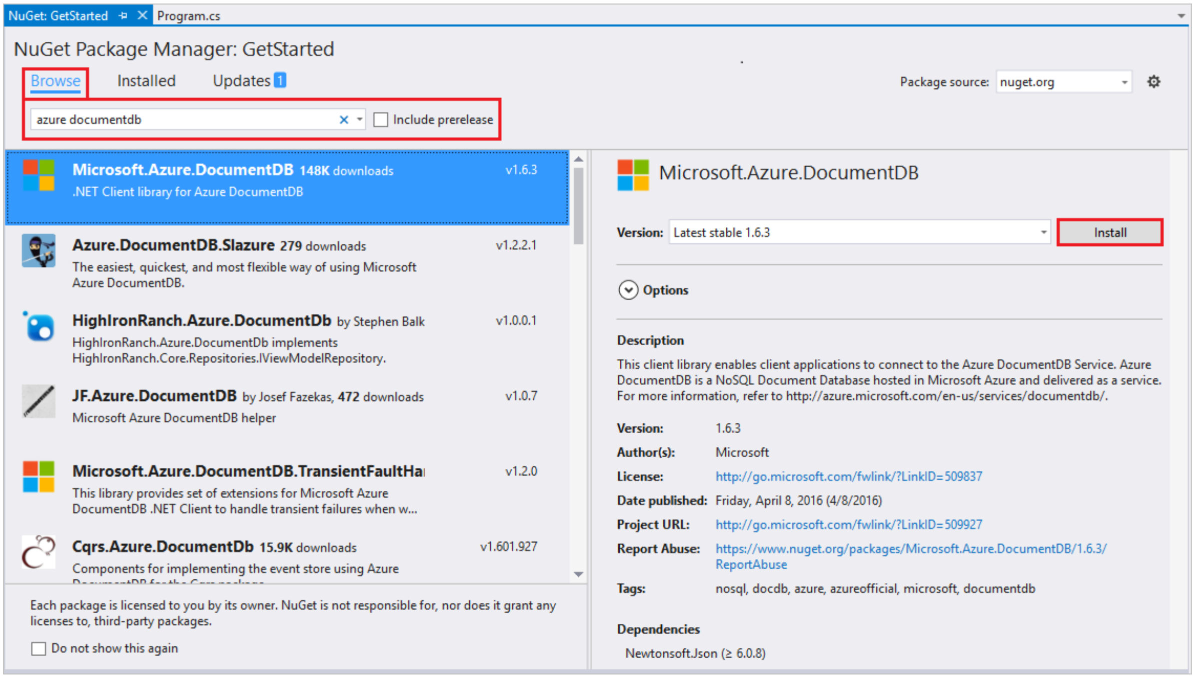
☑ Create directory for solution

□ Add to source control

4. In the **Solution Explorer**, right click on your new console application, which is under your Visual Studio solution, and then click **Manage NuGet Packages...**



5. In the **Nuget** tab, click **Browse**, and type **azure documentdb** in the search box.
6. Within the results, find **Microsoft.Azure.DocumentDB** and click **Install**. The package ID for the Azure Cosmos DB DocumentDB API Client Library is Microsoft Azure DocumentDB Client Library.



Package source: nuget.org

azure documentdb

☐ Include prerelease

Microsoft.Azure.DocumentDB 148K downloads v1.6.3
.NET Client library for Azure DocumentDB

Azure.DocumentDB.Slazure 279 downloads v1.2.2.1
The easiest, quickest, and most flexible way of using Microsoft Azure DocumentDB.

HighIronRanch.Azure.DocumentDb by Stephen Balk v1.0.0.1
HighIronRanch.Azure.DocumentDb implements HighIronRanch.Core.Repositories.IViewModelRepository.

JF.Azure.DocumentDB by Josef Fazekas, 472 downloads v1.0.7
Microsoft Azure DocumentDB helper

Microsoft.Azure.DocumentDB.TransientFaultHa v1.2.0
This library provides set of extensions for Microsoft Azure DocumentDB .NET Client to handle transient failures when w...

Cqrs.Azure.DocumentDb 15.9K downloads v1.601.927
Components for implementing the event store using Azure

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

☐ Do not show this again

Microsoft.Azure.DocumentDB

Version: Latest stable 1.6.3

Description

This client library enables client applications to connect to the Azure DocumentDB Service. Azure DocumentDB is a NoSQL Document Database hosted in Microsoft Azure and delivered as a service. For more information, refer to http://azure.microsoft.com/en-us/services/documentdb/.

Version: 1.6.3
Author(s): Microsoft
License: http://go.microsoft.com/fwlink/?LinkID=509837
Date published: Friday, April 8, 2016 (4/8/2016)
Project URL: http://go.microsoft.com/fwlink/?LinkID=509927
Report Abuse: https://www.nuget.org/packages/Microsoft.Azure.DocumentDB/1.6.3/ReportAbuse
Tags: nosql, docdb, azure, azureofficial, microsoft, documentdb

Dependencies

Newtonsoft.Json (≥ 6.0.8)

If you get a messages about reviewing changes to the solution, click **OK**. If you get a message about license acceptance, click **I accept**.

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 an Azure Cosmos DB 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 the tutorial, make sure you add the dependencies above.

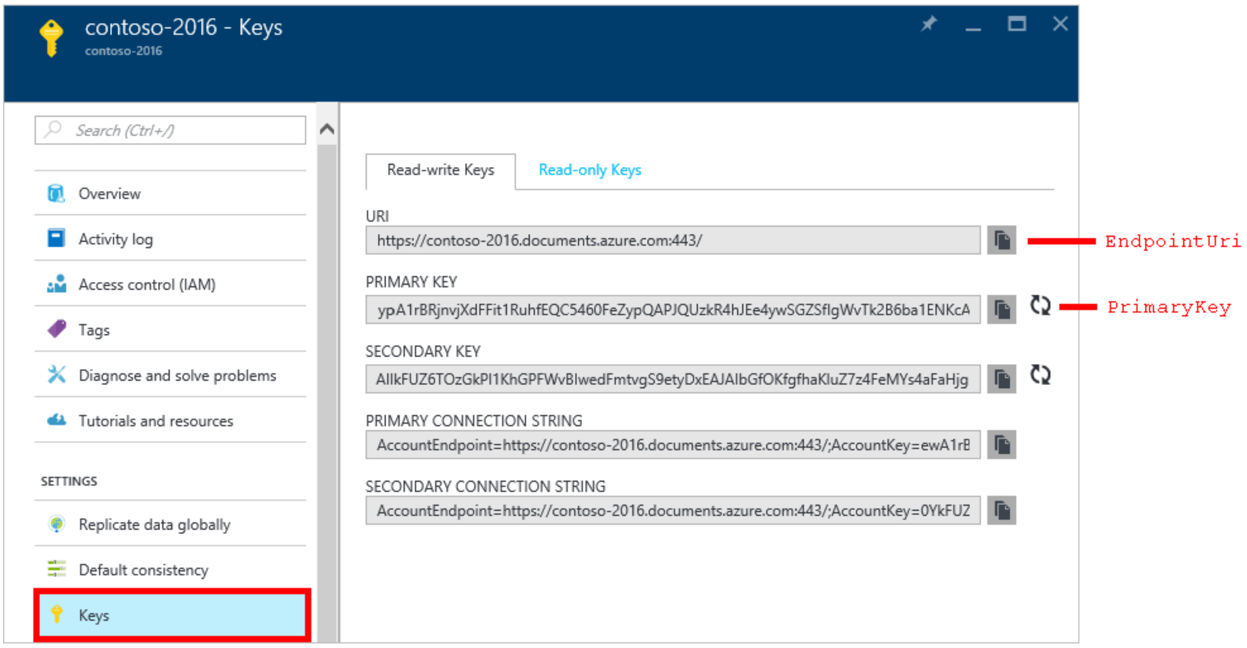
Now, add these two constants and your *client* variable underneath your public class *Program*.

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

Next, head back to the Azure Portal to retrieve your endpoint URL and primary key. The endpoint URL and primary key are necessary for your application to understand where to connect to, and for Azure Cosmos DB to trust your application's connection.

In the Azure Portal, navigate to your Azure Cosmos DB account, and then click **Keys**.

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



URI
https://contoso-2016.documents.azure.com:443/

PRIMARY KEY
ypA1rBRjnvjXdFFit1RuhfEQC5460FeZypQAPJQUzkR4hJEe4ywSGZSflgWvTk2B6ba1ENKcA

SECONDARY KEY
AIlkFUZ6TOzGkPI1KhGPFWvBlwedFmtvgS9etyDxEAJAIbGfOKfgfhaKluZ7z4FeMYs4aFaHjg

PRIMARY CONNECTION STRING
AccountEndpoint=https://contoso-2016.documents.azure.com:443/;AccountKey=ewA1rB

SECONDARY CONNECTION STRING
AccountEndpoint=https://contoso-2016.documents.azure.com:443/;AccountKey=0YkFUZ

Next, we'll start the 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(EndpointUrl), 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. The console window output displays the message `End of demo, press any key to exit.` confirming that the connection was made. You can then close the console window.

Congratulations! You have successfully connected to an Azure Cosmos DB account, let's now take a look at working with Azure Cosmos DB 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 after 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 Azure Cosmos DB database can be created by using the CreateDatabaseIfNotExistsAsync method of the **DocumentClient** class. A database is the logical container of JSON document storage partitioned across collections.

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

```
private async Task GetStartedDemo()
{
  this.client = new DocumentClient(new Uri(EndpointUrl), PrimaryKey);

  // ADD THIS PART TO YOUR CODE
  await this.client.CreateDatabaseIfNotExistsAsync(new Database { Id = "FamilyDB" });
```

Press **F5** to run your application.

Congratulations! You have successfully created an Azure Cosmos DB database.

Step 5: Create a collection

WARNING

CreateDocumentCollectionIfNotExistsAsync 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 CreateDocumentCollectionIfNotExistsAsync method of the **DocumentClient** class. A collection is a container of JSON documents and associated JavaScript application logic.

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

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

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

    // ADD THIS PART TO YOUR CODE
     await this.client.CreateDocumentCollectionIfNotExistsAsync(UriFactory.CreateDatabaseUri("FamilyDB"), new DocumentCollection { Id =
"FamilyCollection" });
```

Press **F5** to run your application.

Congratulations! You have successfully created an Azure Cosmos DB 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 to import the data into a database.

First, we need to create a **Family** class that will represent objects stored within Azure Cosmos DB 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 after 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 **Address** class.

```
// 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.

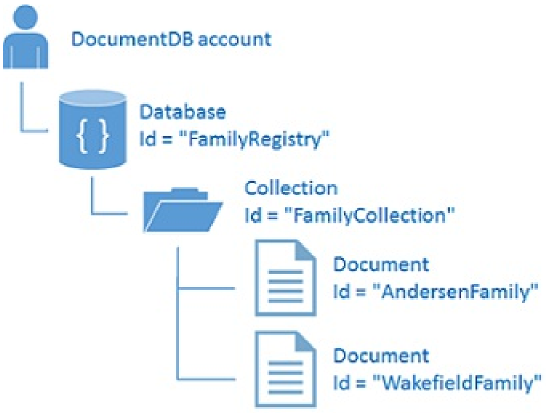
Copy and paste the following code to your **GetStartedDemo** method after the document collection creation.

```
await this.client.CreateDatabaseIfNotExistsAsync(new Database { Id = "FamilyDB" });

await this.client.CreateDocumentCollectionIfNotExistsAsync(UriFactory.CreateDatabaseUri("FamilyDB"), new DocumentCollection { Id =
"FamilyCollection" });


// ADD THIS PART TO YOUR CODE
Family andersenFamily = new Family
{
        Id = "Andersen.1",
        LastName = "Andersen",
        Parents = new Parent[]
        {
                new Parent { FirstName = "Thomas" },
                new Parent { FirstName = "Mary Kay" }
        },
        Children = new Child[]
        {
                new Child
                {
                        FirstName = "Henriette Thaulow",
                        Gender = "female",
                        Grade = 5,
                        Pets = new Pet[]
                        {
                                new Pet { GivenName = "Fluffy" }
                        }
                }
        },
        Address = new Address { State = "WA", County = "King", City = "Seattle" },
        IsRegistered = true
};

await this.CreateFamilyDocumentIfNotExists("FamilyDB", "FamilyCollection", andersenFamily);

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

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

await this.CreateFamilyDocumentIfNotExists("FamilyDB", "FamilyCollection", wakefieldFamily);
```

Press **F5** to run your application.

Congratulations! You have successfully created two Azure Cosmos DB documents.



Step 7: Query Azure Cosmos DB resources

Azure Cosmos DB supports rich queries against JSON documents stored in each collection. The following sample code shows various queries - using both Azure Cosmos DB 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 after 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();
}
```

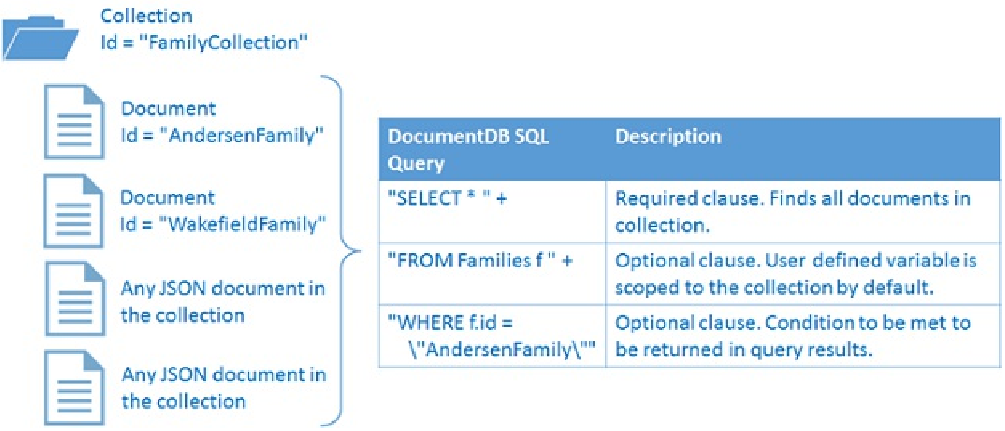
Copy and paste the following code to your **GetStartedDemo** method after the second document creation.

```
await this.CreateFamilyDocumentIfNotExists("FamilyDB", "FamilyCollection", wakefieldFamily);

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

Press **F5** to run your application.

Congratulations! You have successfully queried against an Azure Cosmos DB collection.

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



DocumentDB SQL Query	Description
"SELECT * " +	Required clause. Finds all documents in collection.
"FROM Families f " +	Optional clause. User defined variable is scoped to the collection by default.
"WHERE f.id = \"AndersenFamily\""	Optional clause. Condition to be met to be returned in query results.

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

Azure Cosmos DB supports replacing JSON documents.

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

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

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

```
await this.CreateFamilyDocumentIfNotExists("FamilyDB", "FamilyCollection", wakefieldFamily);

this.ExecuteSimpleQuery("FamilyDB", "FamilyCollection");

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

await this.ReplaceFamilyDocument("FamilyDB", "FamilyCollection", "Andersen.1", andersenFamily);

this.ExecuteSimpleQuery("FamilyDB", "FamilyCollection");
```

Press **F5** to run your application.

Congratulations! You have successfully replaced an Azure Cosmos DB document.

Step 9: Delete JSON document

Azure Cosmos DB supports deleting JSON documents.

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

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

Copy and paste the following code to your **GetStartedDemo** method after the second query execution, at the end of the method.

```
await this.ReplaceFamilyDocument("FamilyDB", "FamilyCollection", "Andersen.1", andersenFamily);

this.ExecuteSimpleQuery("FamilyDB", "FamilyCollection");

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

Press **F5** to run your application.

Congratulations! You have successfully deleted an Azure Cosmos DB 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 after the document delete to delete the entire database and all children resources.

```
this.ExecuteSimpleQuery("FamilyDB", "FamilyCollection");

await this.DeleteFamilyDocument("FamilyDB", "FamilyCollection", "Andersen.1");

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

Press **F5** to run your application.

Congratulations! You have successfully deleted an Azure Cosmos DB database.

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

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

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

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

Congratulations! You've completed the tutorial and have a working C# console application!

Get the complete tutorial solution

If you didn't have time to complete the steps in this tutorial, or just want to download the code samples, you can get it from GitHub.

To build the GetStarted solution, you will need the following:

- An active Azure account. If you don't have one, you can sign up for a free account.
- A Azure Cosmos DB 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 an Azure Cosmos DB account.

That's it, build it and you're on your way!

Next steps

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

Azure Cosmos DB: Getting started with the DocumentDB API and .NET Core

6/6/2017 • 15 min to read • Edit Online

Welcome to the Azure Cosmos DB getting started tutorial! After following this tutorial, you'll have a console application that creates and queries DocumentDB resources.

We'll cover:

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

Want to build a Xamarin iOS, Android, or Forms application using the DocumentDB .NET Core SDK? See Developing Xamarin mobile applications using DocumentDB.

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.

> NOTE
>
> The DocumentDB .NET Core SDK used in this tutorial is not yet compatible with Universal Windows Platform (UWP) apps. For a preview version of the .NET Core SDK that does support UWP apps, send email to askcosmosdb@microsoft.com.

Now let's get started!

Prerequisites

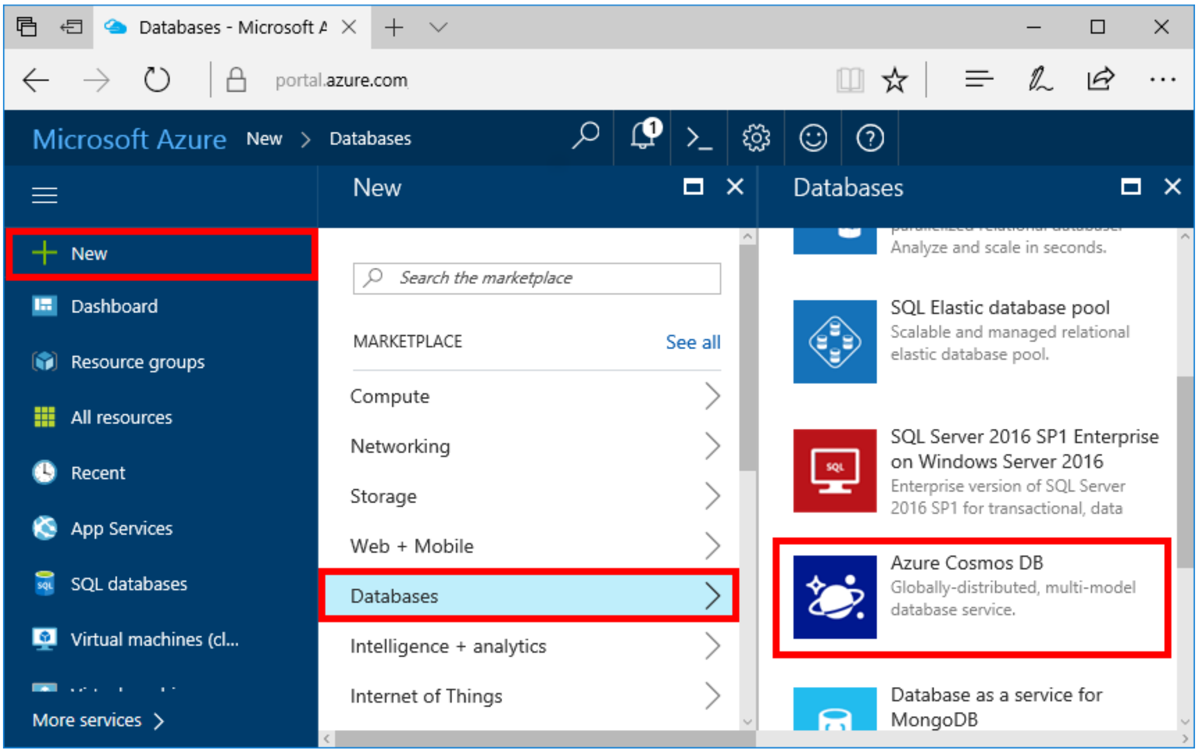
Please make sure you have the following:

- An active Azure account. If you don't have one, you can sign up for a free account.
  - Alternatively, you can use the Azure Cosmos DB Emulator for this tutorial.
- Visual Studio 2017
  - 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 an Azure Cosmos DB 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 Azure Cosmos DB Emulator, please follow the steps at Azure Cosmos DB Emulator to setup the emulator and skip ahead to Setup your Visual Studio Solution.

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

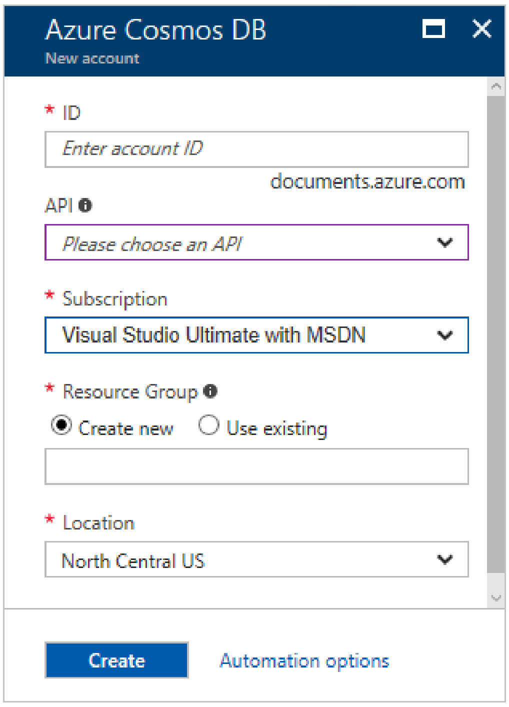


3. On the **New account** blade, specify the configuration that you want for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article we program against the DocumentDB API, so choose **SQL (DocumentDB)** as you fill out the form. But if you have graph data for a social media app, or key/value (table) data, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

   Complete the fields on the **New account** blade, using the information in the following screenshot as a guide. When you set up your account, choose unique values that do not match those in the screenshot.



Azure Cosmos DB
New account

* ID

Enter account ID

documents.azure.com

API

Please choose an API

* Subscription

Visual Studio Ultimate with MSDN

* Resource Group

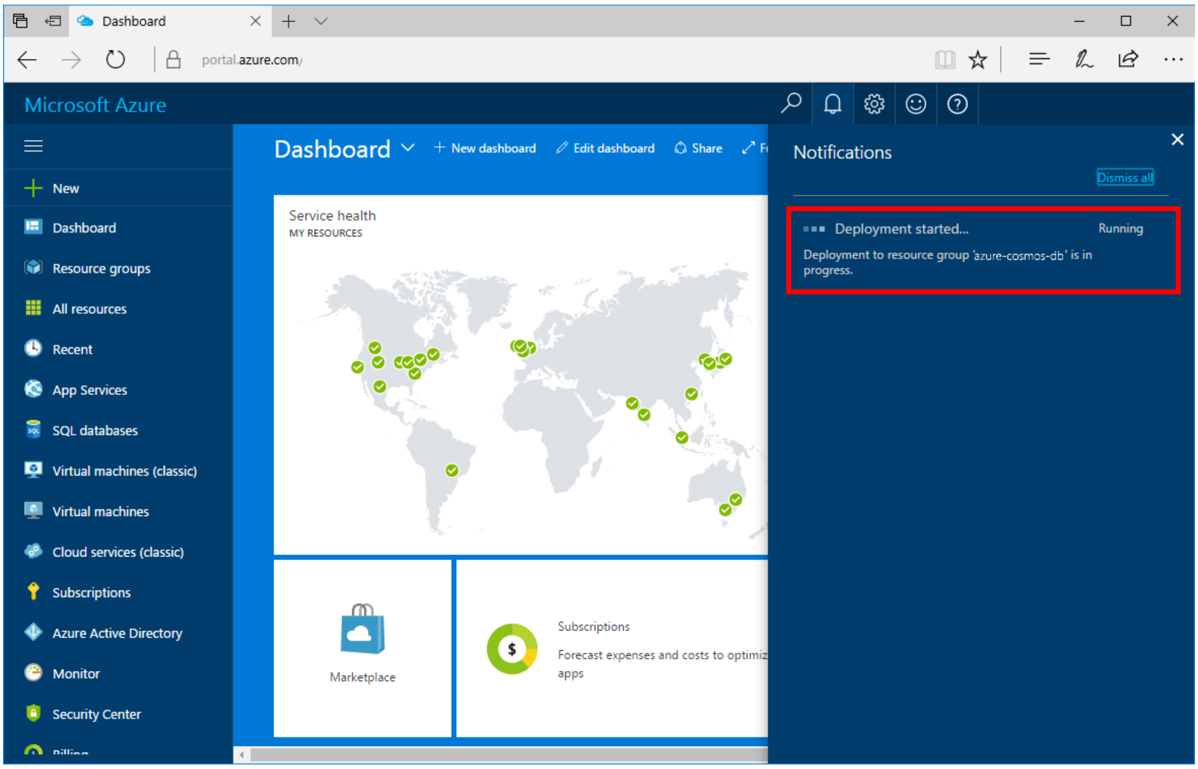
● Create new ○ Use existing

* Location

North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that identifies your Azure Cosmos DB account. The string *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 through 50 characters.
API	SQL (DocumentDB)	We program against the DocumentDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for your Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource-group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location that's closest to your users to give them the fastest access to the data.

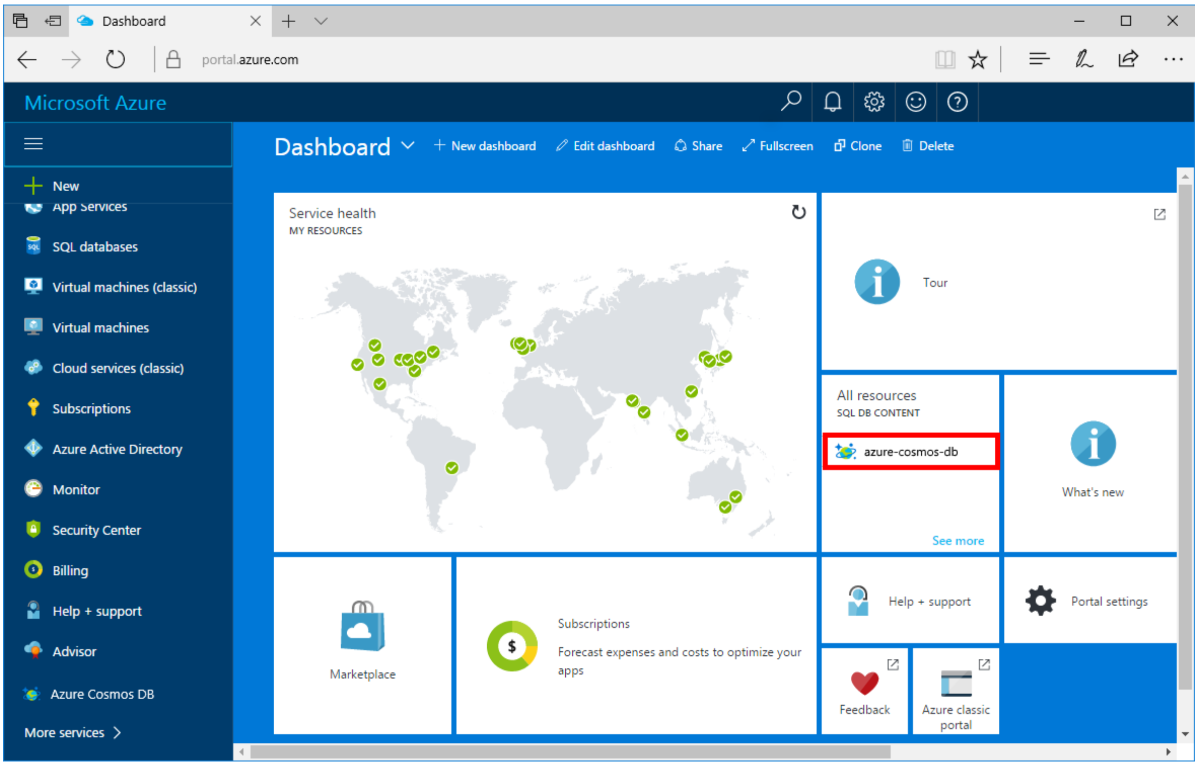
4. Click **Create** to create the account.
5. On the top toolbar, click **Notifications** to monitor the deployment process.



Deployment started...

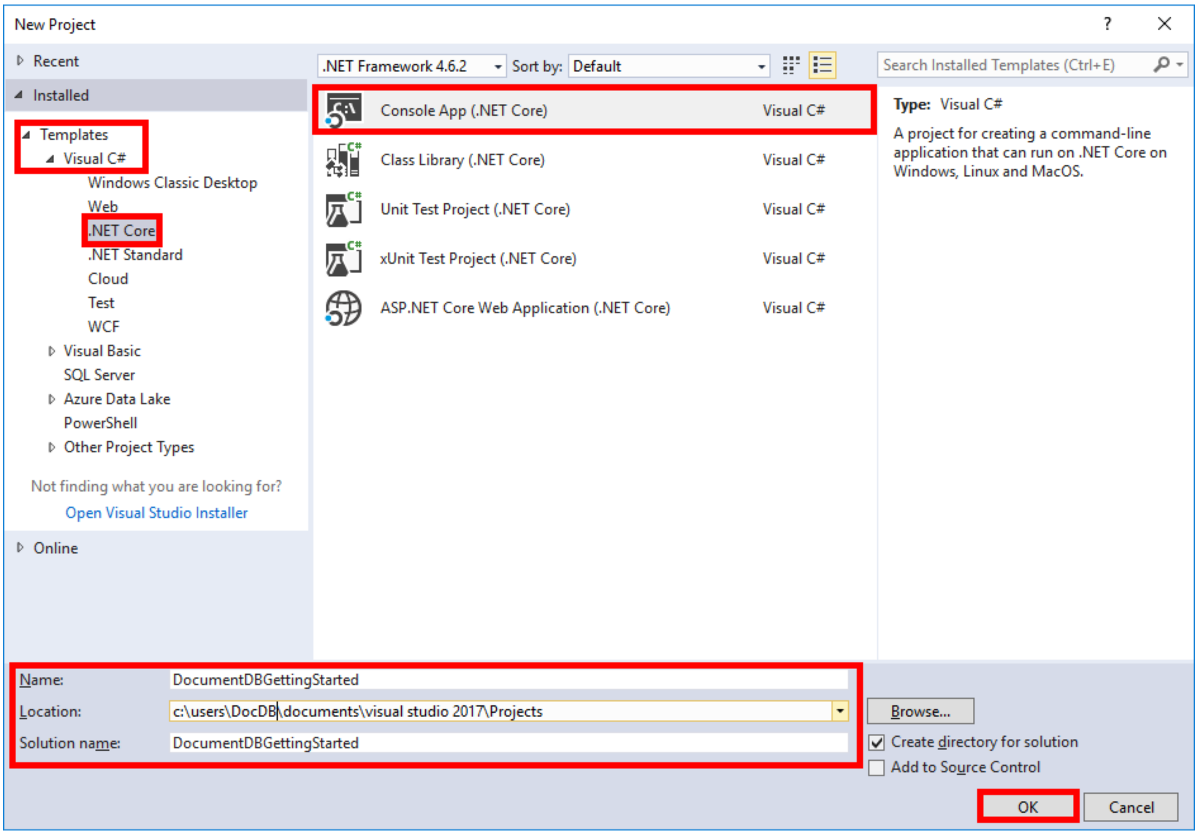
Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the **All Resources** tile.



Step 2: Setup your Visual Studio solution

1. Open **Visual Studio 2017** 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 **DocumentDBGettingStarted**, and then click **OK**.


New Project
Recent
Installed
Templates
Visual C#
Windows Classic Desktop
Web
.NET Core
.NET Standard
Cloud
Test
WCF
Visual Basic
SQL Server
Azure Data Lake
PowerShell
Other Project Types
Not finding what you are looking for?
Open Visual Studio Installer
Online
.NET Framework 4.6.2
Sort by: Default
Search Installed Templates (Ctrl+E)
Console App (.NET Core) Visual C#
Class Library (.NET Core) Visual C#
Unit Test Project (.NET Core) Visual C#
xUnit Test Project (.NET Core) Visual C#
ASP.NET Core Web Application (.NET Core) Visual C#
Browse...
OK
Cancel

Type: Visual C#

A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.

Name: DocumentDBGettingStarted

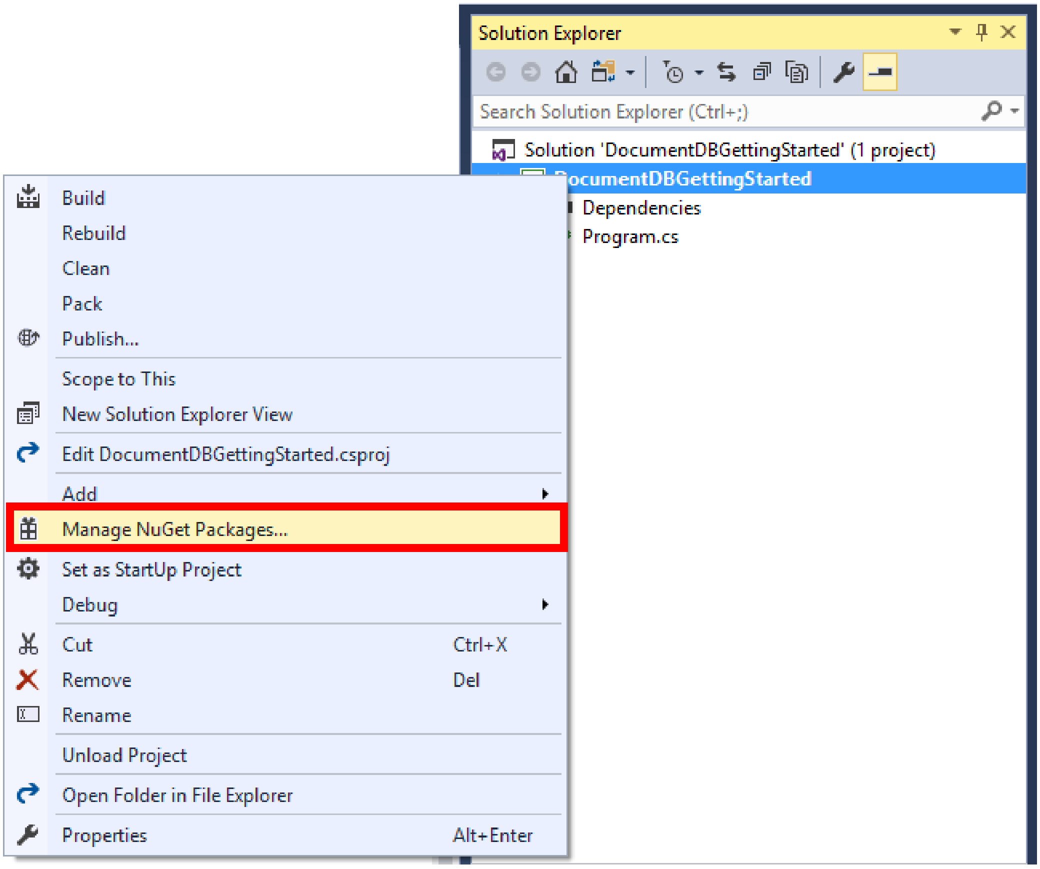
Location: c:\users\DocDB\documents\visual studio 2017\Projects

Solution name: DocumentDBGettingStarted

☑ Create directory for solution

□ Add to Source Control

4. In the **Solution Explorer**, right click on **DocumentDBGettingStarted**.
5. Then without leaving the menu, click on **Manage NuGet Packages...**.



6. In the **NuGet** tab, click **Browse** at the top of the window, 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 for .NET Core is Microsoft.Azure.DocumentDB.Core. If you are targeting a .NET Framework version (like net461) that is not supported by this .NET Core NuGet package, then use Microsoft.Azure.DocumentDB that supports all .NET Framework versions starting .NET Framework 4.5.

8. At the prompts, accept the NuGet package installations and the license agreement.

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 an Azure Cosmos DB account

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

```
using System;

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

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

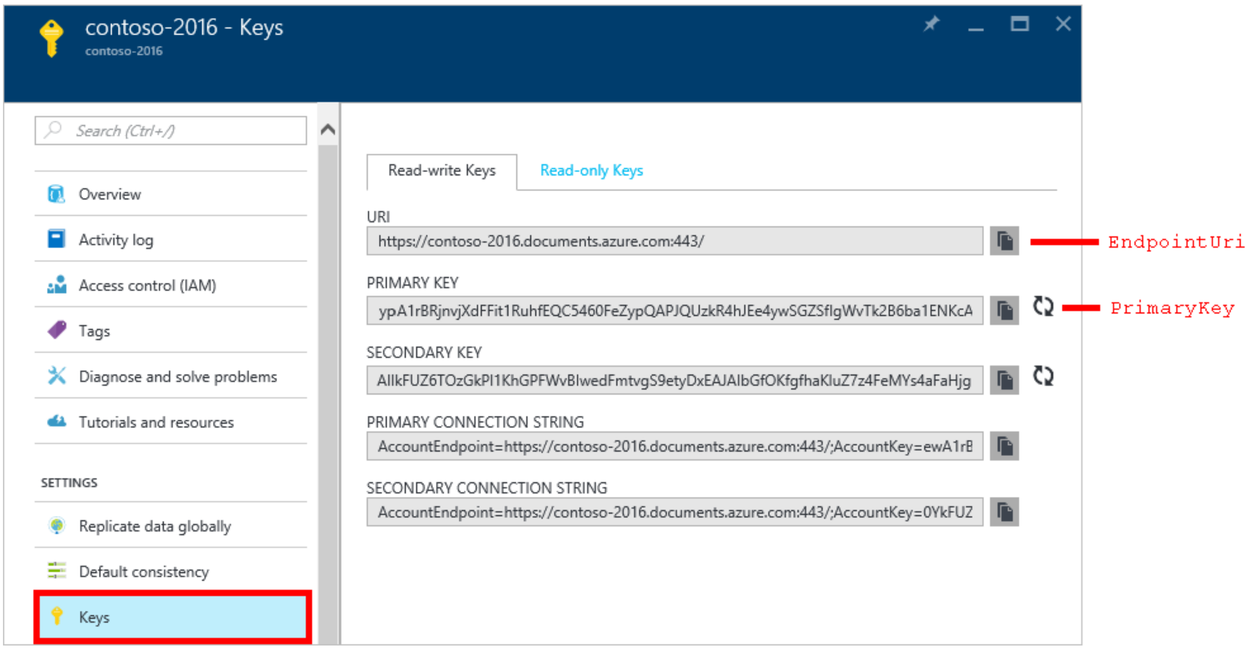
Now, add these two constants and your *client* variable underneath your public class *Program*.

```
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 Azure Cosmos DB 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 Cosmos DB Emulator, use `https://localhost:8081` as the endpoint, and the well-defined authorization key from How to develop using the Azure Cosmos DB Emulator. Make sure to remove the < and > but leave the double quotes around your endpoint and key.



URI
https://contoso-2016.documents.azure.com:443/

PRIMARY KEY
ypA1rBRjnvjXdFFit1RuhfEQC5460FeZypQAPJQUzkR4hJEe4ywSGZSflgWvTk2B6ba1ENKcA

SECONDARY KEY
AllkFUZ6TOzGkPl1KhGPFWvBlwedFmtvgS9etyDxEAJAlbGfOKfgfhaKluZ7z4FeMYs4aFaHjg

PRIMARY CONNECTION STRING
AccountEndpoint=https://contoso-2016.documents.azure.com:443/;AccountKey=ewA1rB

SECONDARY CONNECTION STRING
AccountEndpoint=https://contoso-2016.documents.azure.com:443/;AccountKey=0YkFUZ

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 the **DocumentDBGettingStarted** button to build and run the application.

Congratulations! You have successfully connected to an Azure Cosmos DB account, let's now take a look at working with Azure Cosmos DB 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 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.client.CreateDatabaseIfNotExistsAsync(new Database { Id = "FamilyDB_oa" });
```

Press the **DocumentDBGettingStarted** button to run your application.

Congratulations! You have successfully created an Azure Cosmos DB 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 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.client.CreateDocumentCollectionIfNotExistsAsync(UriFactory.CreateDatabaseUri("FamilyDB_oa"), new DocumentCollection { Id =
 "FamilyCollection_oa" });
```

Press the **DocumentDBGettingStarted** button to run your application.

Congratulations! You have successfully created an Azure Cosmos DB 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 Azure Cosmos DB 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;
    }
  }
}
```

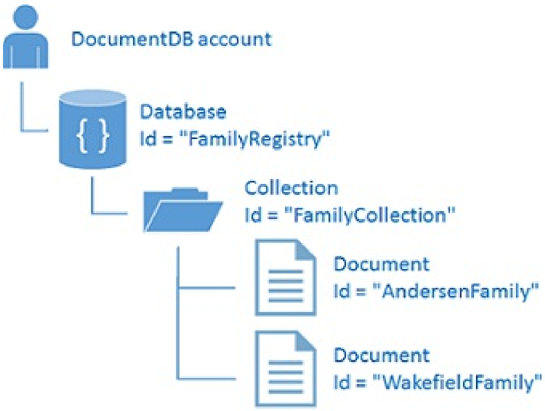
And insert two documents, one each for the Andersen Family and the Wakefield Family.

Copy and paste the code that follows `// ADD THIS PART TO YOUR 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[]
    {
        {
            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 the **DocumentDBGettingStarted** button to run your application.

Congratulations! You have successfully created two Azure Cosmos DB documents.



Step 7: Query Azure Cosmos DB resources

Azure Cosmos DB supports rich queries against JSON documents stored in each collection. The following sample code shows various queries - using both Azure Cosmos DB 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();
}
```

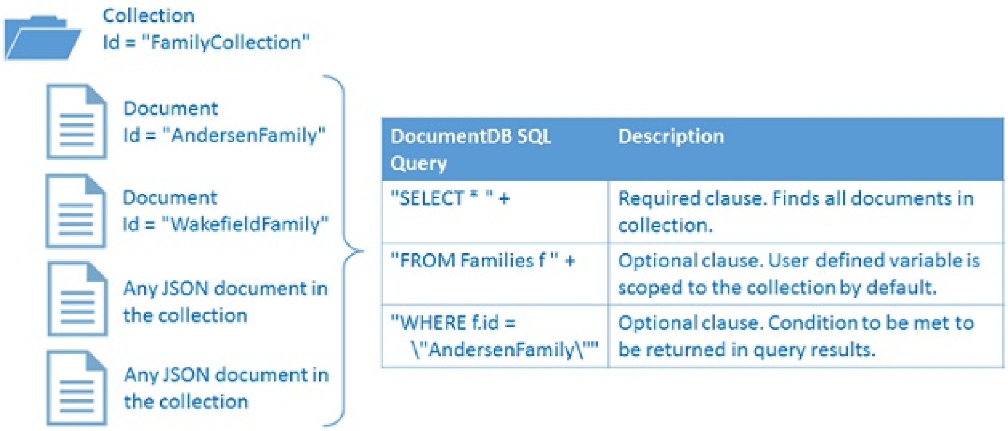
Copy and paste the following code to your **GetStartedDemo** method underneath the second document creation.

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

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

Press the **DocumentDBGettingStarted** button to run your application.

Congratulations! You have successfully queried against an Azure Cosmos DB collection.

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



DocumentDB SQL Query	Description
"SELECT * " +	Required clause. Finds all documents in collection.
"FROM Families f " +	Optional clause. User defined variable is scoped to the collection by default.
"WHERE f.id = \"AndersenFamily\""	Optional clause. Condition to be met to be returned in query results.

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

Azure Cosmos DB 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 the **DocumentDBGettingStarted** button to run your application.

Congratulations! You have successfully replaced a DocumentDB document.

Step 9: Delete JSON document

Azure Cosmos DB 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 the **DocumentDBGettingStarted** button to run your application.

Congratulations! You have successfully deleted an Azure Cosmos DB 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 the **DocumentDBGettingStarted** button to run your application.

Congratulations! You have successfully deleted an Azure Cosmos DB database.

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

Press the **DocumentDBGettingStarted** button 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 the tutorial and have a working C# console application!

Get the complete 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 Azure Cosmos DB 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 tutorial? See Build a web application with ASP.NET MVC using DocumentDB.
- Want to develop a Xamarin iOS, Android, or Forms application using the DocumentDB .NET Core SDK? See Developing Xamarin mobile applications using DocumentDB.
- Want to perform scale and performance testing with Azure Cosmos DB? See Performance and Scale Testing with Azure Cosmos DB
- Learn how to monitor an Azure Cosmos DB 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 Java console application

6/9/2017 • 6 min to read • Edit Online

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

We cover:

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

Now let's get started!

Prerequisites

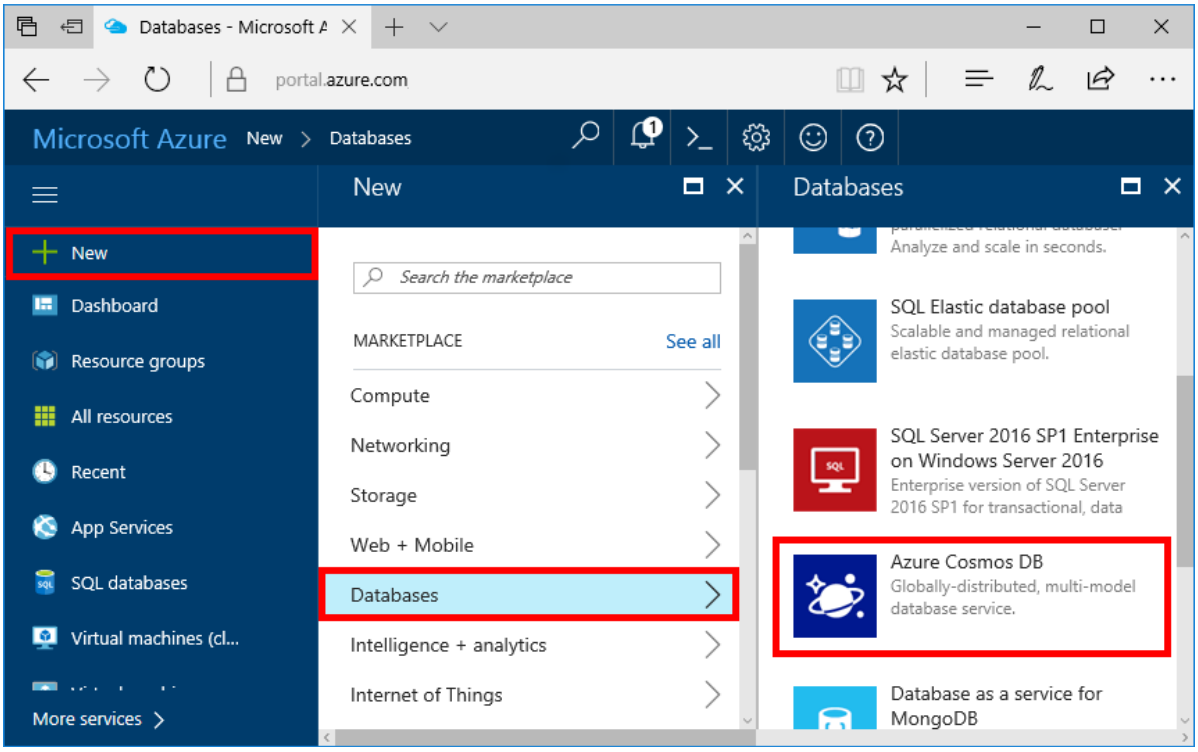
Make sure you have the following:

- An active Azure account. If you don't have one, you can sign up for a free account. Alternatively, you can use the Azure Cosmos DB Emulator for this tutorial.
- Git
- Java Development Kit (JDK) 7+.
- Maven.

Step 1: Create an Azure Cosmos DB account

Let's create an Azure Cosmos DB account. If you already have an account you want to use, you can skip ahead to Clone the GitHub project. If you are using the Azure Cosmos DB Emulator, follow the steps at Azure Cosmos DB Emulator to set up the emulator and skip ahead to Clone the GitHub project.

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

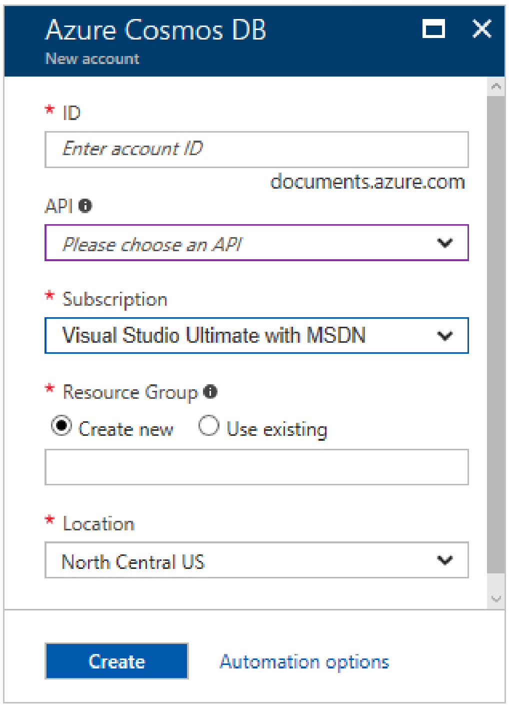


3. On the **New account** blade, specify the configuration that you want for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article we program against the DocumentDB API, so choose **SQL (DocumentDB)** as you fill out the form. But if you have graph data for a social media app, or key/value (table) data, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

   Complete the fields on the **New account** blade, using the information in the following screenshot as a guide. When you set up your account, choose unique values that do not match those in the screenshot.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

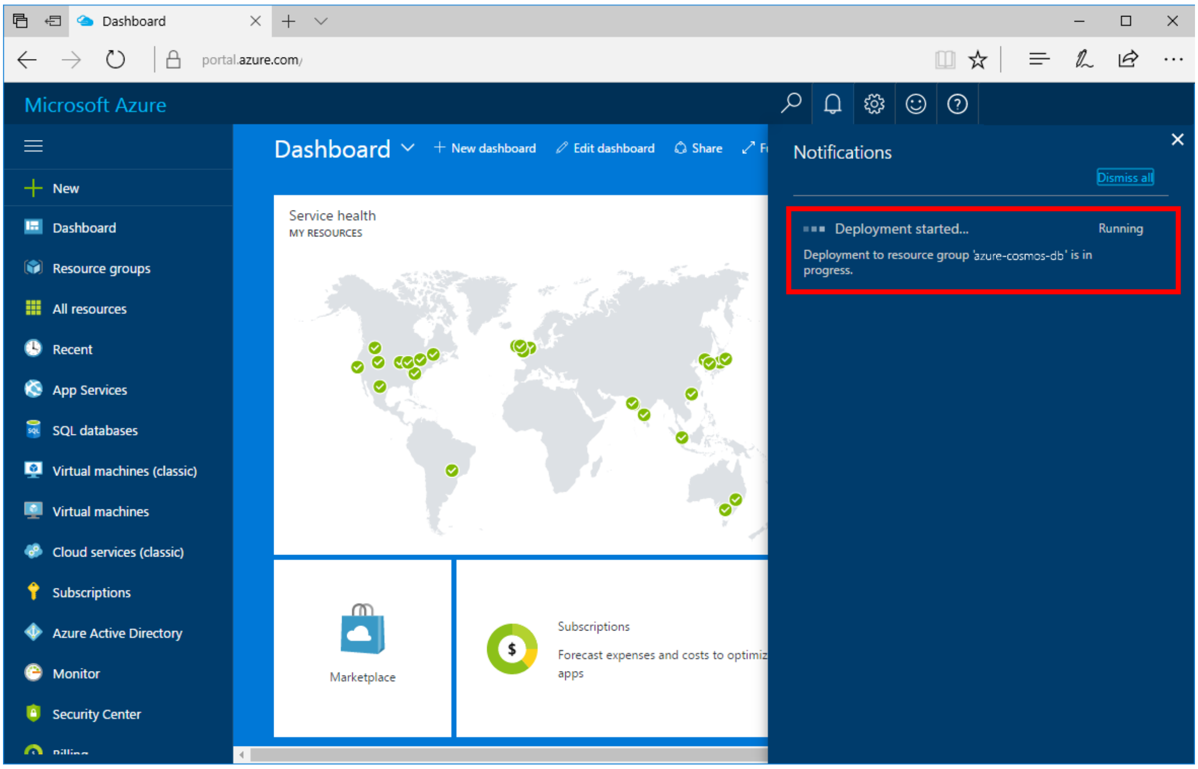
* Subscription
Visual Studio Ultimate with MSDN

* Resource Group
● Create new ○ Use existing

* Location
North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that identifies your Azure Cosmos DB account. The string *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 through 50 characters.
API	SQL (DocumentDB)	We program against the DocumentDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for your Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource-group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location that's closest to your users to give them the fastest access to the data.

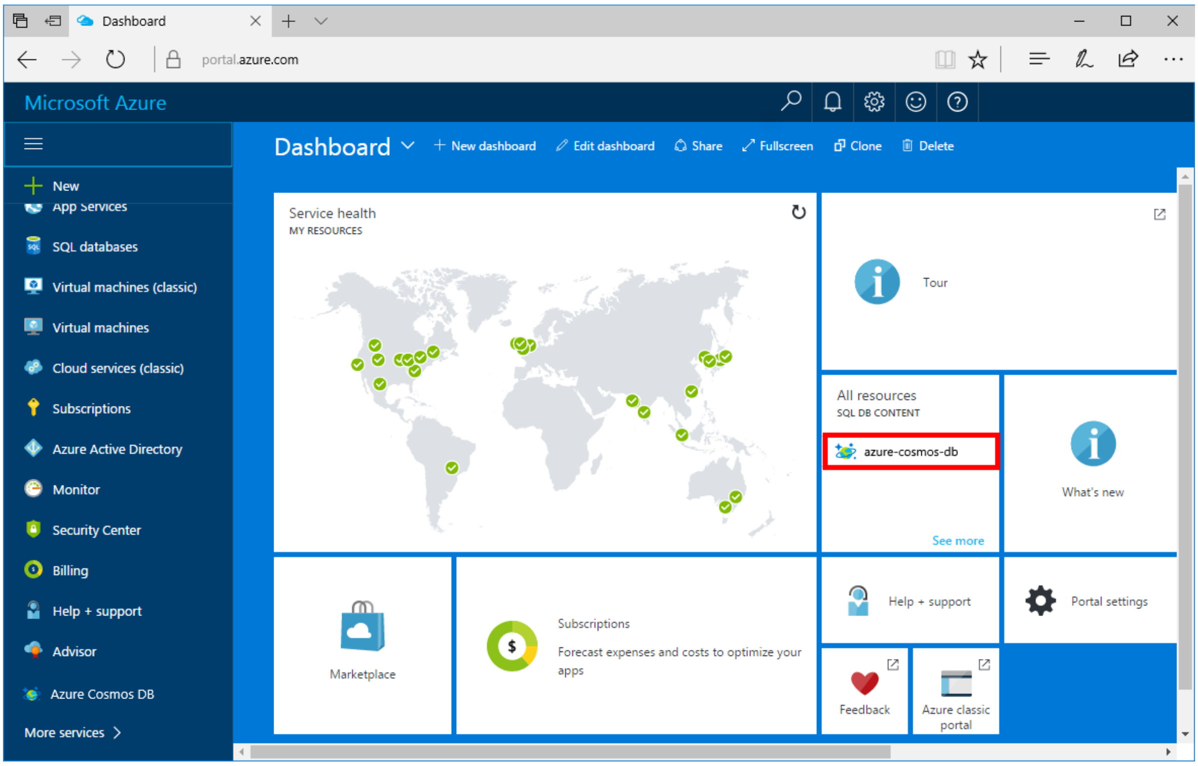
4. Click **Create** to create the account.
5. On the top toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the **All Resources** tile.



Step 2: Clone the GitHub project

You can get started by cloning the GitHub repository for Get Started with Azure Cosmos DB and Java. For example, from a local directory run the following to retrieve the sample project locally.

```
git clone git@github.com:Azure-Samples/azure-cosmos-db-documentdb-java-getting-started.git

cd azure-cosmos-db-documentdb-java-getting-started
```

The directory contains a `pom.xml` for the project and a `src` folder containing Java source code including `Program.java` which shows how perform simple operations with Azure DocumentDB like creating documents and querying data within a collection. The `pom.xml` includes a dependency on the DocumentDB Java SDK on Maven.

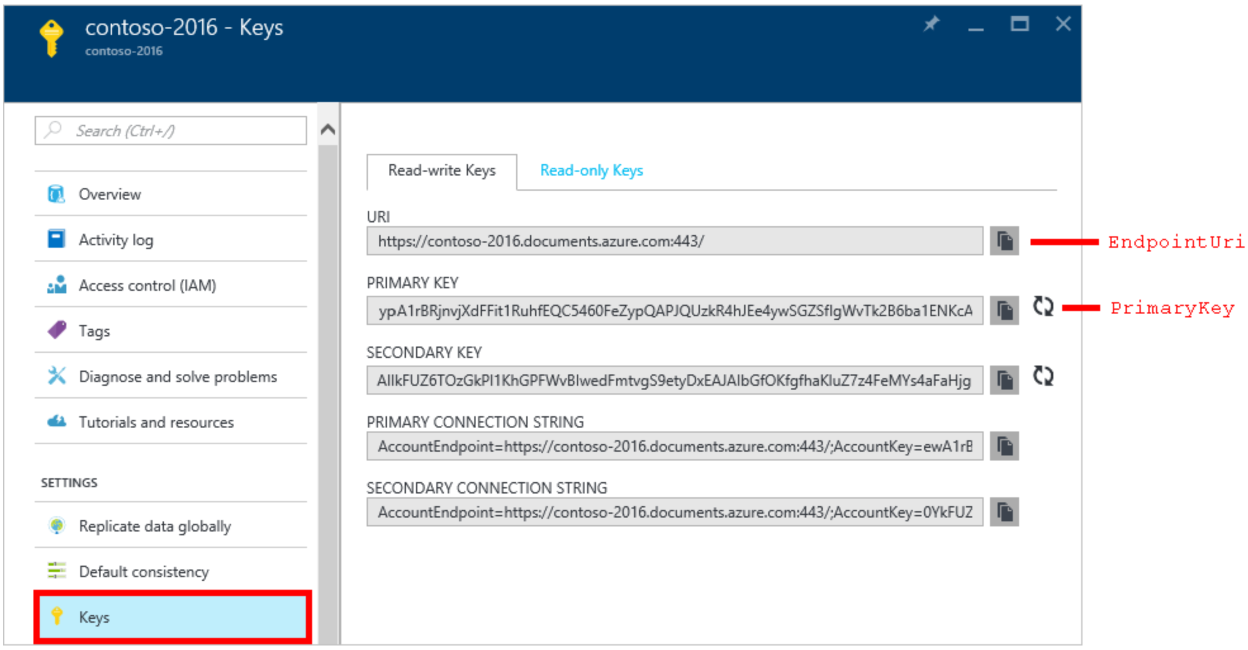
```
<dependency>
    <groupId>com.microsoft.azure</groupId>
    <artifactId>azure-documentdb</artifactId>
    <version>LATEST</version>
</dependency>
```

Step 3: Connect to an Azure Cosmos DB account

Next, head back to the Azure Portal to retrieve your endpoint and primary master key. The Azure Cosmos DB endpoint and primary key are necessary for your application to understand where to connect to, and for Azure Cosmos DB to trust your application's connection.

In the Azure Portal, navigate to your Azure Cosmos DB account, and then click **Keys**. Copy the URI from the portal and paste it into `<your endpoint URI>` in the Program.java file. Then copy the PRIMARY KEY from the portal and paste it into `<your key>`.

```
this.client = new DocumentClient(
   "<your endpoint URI>",
   "<your key>"
   , new ConnectionPolicy(),
   ConsistencyLevel.Session);
```



URI
https://contoso-2016.documents.azure.com:443/

PRIMARY KEY
ypA1rBRjnvjXdFFit1RuhfEQC5460FeZypQAPJQUzkR4hJEe4ywSGZSflgWvTk2B6ba1ENKcA

SECONDARY KEY
AIlkFUZ6TOzGkPI1KhGPFWvBlwedFmtvgS9etyDxEAJAIbGfOKfgfhaKluZ7z4FeMYs4aFaHjg

PRIMARY CONNECTION STRING
AccountEndpoint=https://contoso-2016.documents.azure.com:443/;AccountKey=ewA1rB

SECONDARY CONNECTION STRING
AccountEndpoint=https://contoso-2016.documents.azure.com:443/;AccountKey=0YkFUZ

Step 4: Create a database

Your Azure Cosmos DB database can be created by using the createDatabase method of the **DocumentClient** class. A database is the logical container of JSON document storage partitioned across collections.

```
Database database = new Database();
database.setId("familydb");
this.client.createDatabase(database, null);
```

Step 5: Create a collection

WARNING

createCollection creates a new collection with reserved throughput, which has pricing implications. For more details, visit our pricing page.

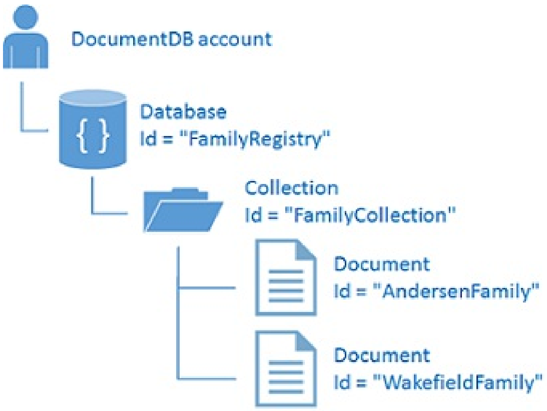
A collection can be created by using the createCollection method of the **DocumentClient** class. A collection is a container of JSON documents and associated JavaScript application logic.

```
DocumentCollection collectionInfo = new DocumentCollection();
collectionInfo.setId("familycoll");

// Azure Cosmos DB collections can be reserved with throughput specified in request units/second.
// Here we create a collection with 400 RU/s.
RequestOptions requestOptions = new RequestOptions();
requestOptions.setOfferThroughput(400);

this.client.createCollection("/dbs/familydb", collectionInfo, requestOptions);
```

Step 6: Create JSON documents

A document can be created by using the createDocument 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 to import the data into a database.

```
// Insert your Java objects as documents
Family andersenFamily = new Family();
andersenFamily.setId("Andersen.1");
andersenFamily.setLastName("Andersen")

// More initialization skipped for brevity. You can have nested references
andersenFamily.setParents(new Parent[] { parent1, parent2 });
andersenFamily.setDistrict("WA5");
Address address = new Address();
address.setCity("Seattle");
address.setCounty("King");
address.setState("WA");

andersenFamily.setAddress(address);
andersenFamily.setRegistered(true);

this.client.createDocument("/dbs/familydb/colls/familycoll", family, new RequestOptions(), true);
```



Step 7: Query Azure Cosmos DB resources

Azure Cosmos DB supports rich queries against JSON documents stored in each collection. The following sample code shows how to query documents in Azure Cosmos DB using SQL syntax with the queryDocuments method.

```
FeedResponse<Document> queryResults = this.client.queryDocuments(
    "/dbs/familydb/colls/familycoll",
    "SELECT * FROM Family WHERE Family.lastName = 'Andersen'",
    null);

System.out.println("Running SQL query...");
for (Document family : queryResults.getQueryIterable()) {
    System.out.println(String.format("\tRead %s", family));
}
```

Step 8: Replace JSON document

Azure Cosmos DB supports updating JSON documents using the replaceDocument method.

```
// Update a property
andersenFamily.Children[0].Grade = 6;

this.client.replaceDocument(
    "/dbs/familydb/colls/familycoll/docs/Andersen.1",
    andersenFamily,
    null);
```

Step 9: Delete JSON document

Similarly, Azure Cosmos DB supports deleting JSON documents using the deleteDocument method.

```
this.client.delete("/dbs/familydb/colls/familycoll/docs/Andersen.1", null);
```

Step 10: Delete the database

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

```
this.client.deleteDatabase("/dbs/familydb", null);
```

Step 11: Run your Java console application all together!

To run the application from the console, navigate to the project folder and compile using Maven:

```
mvn package
```

Running `mvn package` downloads the latest Azure Cosmos DB library from Maven and produces `GetStarted-0.0.1-SNAPSHOT.jar`. Then run the app by running:

```
mvn exec:java -D exec.mainClass=GetStarted.Program
```

Congratulations! You've completed this NoSQL tutorial and have a working Java console application!

Next steps

- Want a Java web app tutorial? See Build a web application with Java using Azure Cosmos DB.
- Learn how to monitor an Azure Cosmos DB account.
- Run queries against our sample dataset in the Query Playground.
- Learn more about the programming model in the Develop section of the Azure Cosmos DB documentation page.

Node.js tutorial: DocumentDB Node.js console application

6/6/2017 • 14 min to read • Edit Online

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

We'll cover:

- Creating and connecting to an Azure Cosmos DB 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

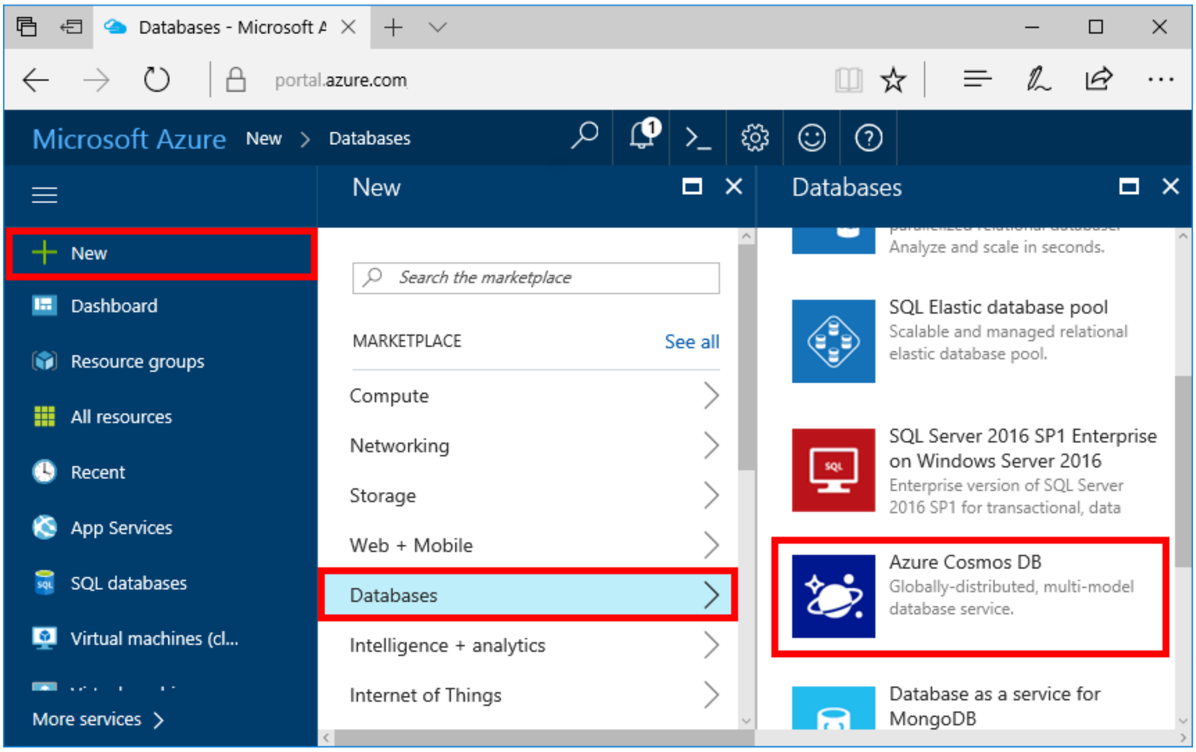
Please make sure you have the following:

- 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 Cosmos DB Emulator for this tutorial.
- Node.js version v0.10.29 or higher.

Step 1: Create an Azure Cosmos DB account

Let's create an Azure Cosmos DB 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 Azure Cosmos DB Emulator, please follow the steps at Azure Cosmos DB Emulator to setup the emulator and skip ahead to Setup your Node.js application.

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

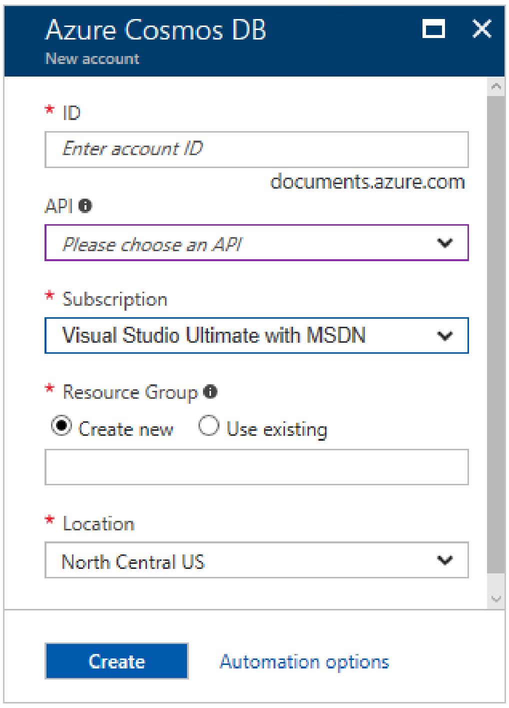


3. On the **New account** blade, specify the configuration that you want for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article we program against the DocumentDB API, so choose **SQL (DocumentDB)** as you fill out the form. But if you have graph data for a social media app, or key/value (table) data, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

   Complete the fields on the **New account** blade, using the information in the following screenshot as a guide. When you set up your account, choose unique values that do not match those in the screenshot.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

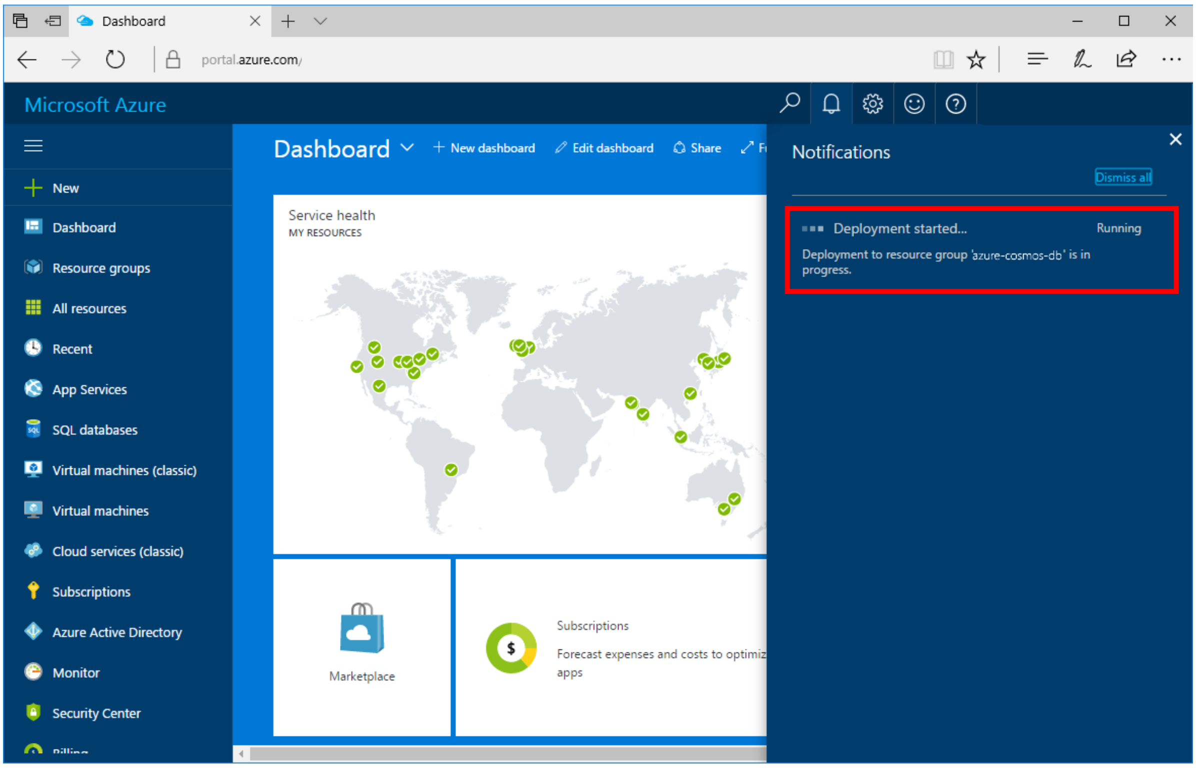
* Subscription
Visual Studio Ultimate with MSDN

* Resource Group
● Create new ○ Use existing

* Location
North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that identifies your Azure Cosmos DB account. The string *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 through 50 characters.
API	SQL (DocumentDB)	We program against the DocumentDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for your Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource-group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location that's closest to your users to give them the fastest access to the data.

4. Click **Create** to create the account.
5. On the top toolbar, click **Notifications** to monitor the deployment process.



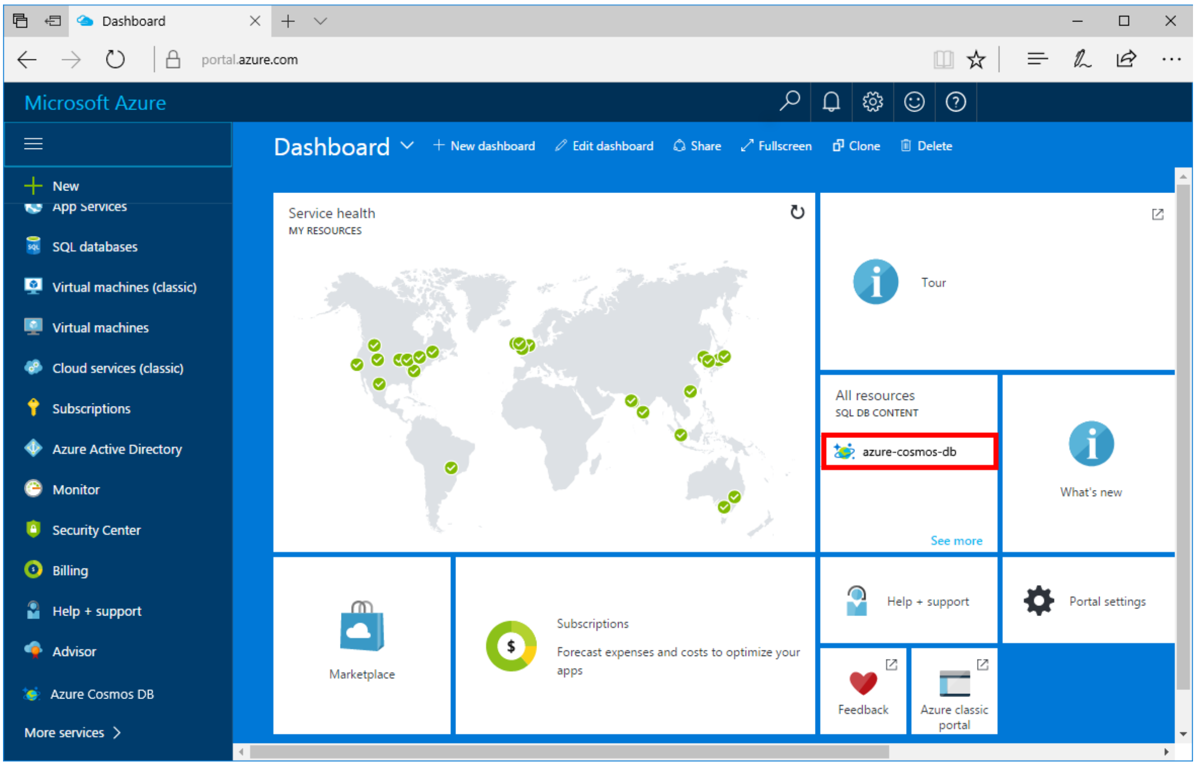
Notifications

Dismiss all

Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the **All Resources** tile.



Step 2: Setup your Node.js application

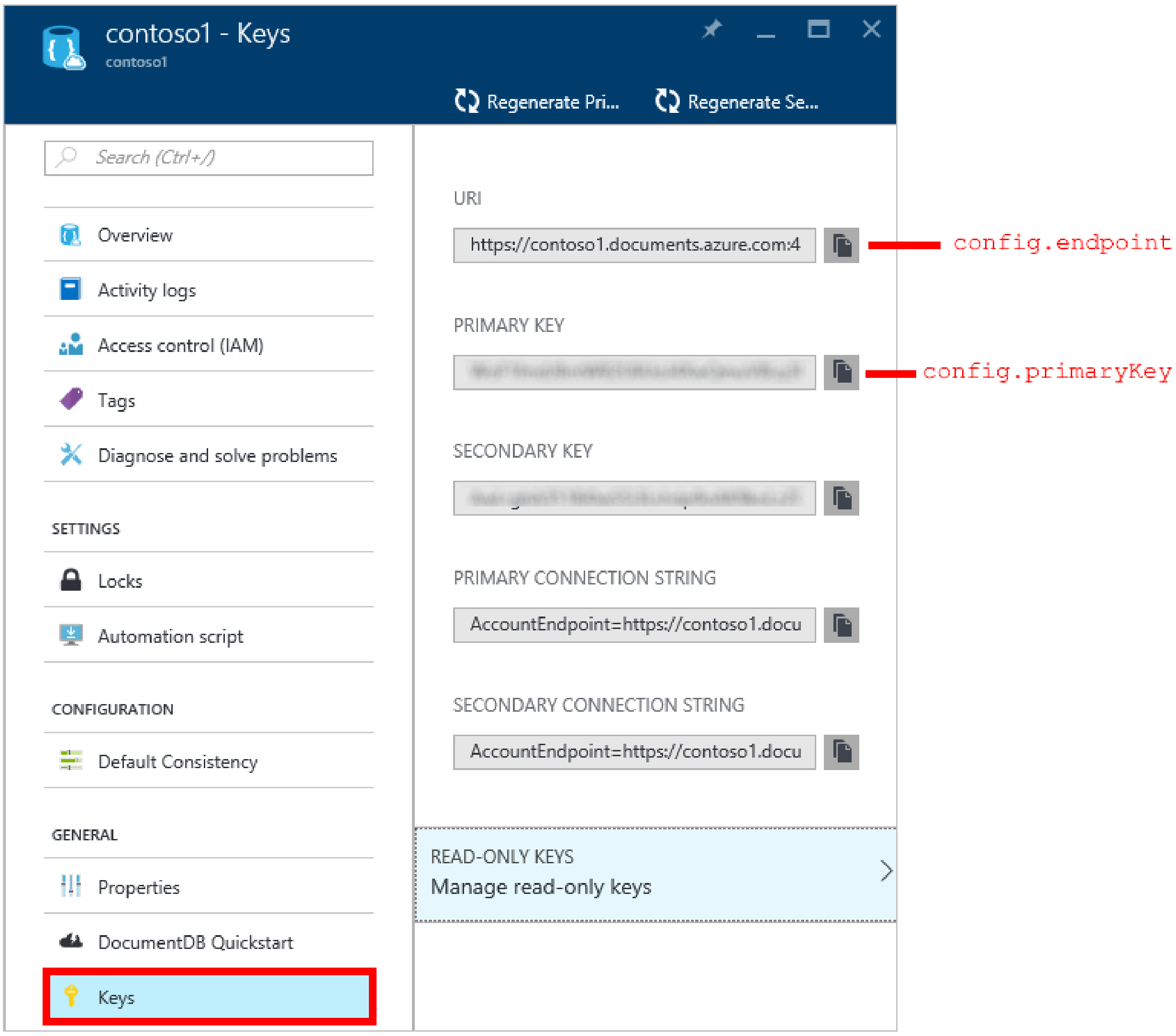
1. Open your favorite terminal.
2. Locate the folder or directory where you'd like to save your Node.js application.
3. Create two empty JavaScript files with the following commands:
   - Windows:
     - `fsutil file createnew app.js 0`
     - `fsutil file createnew config.js 0`
   - Linux/OS X:
   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



URI

https://contoso1.documents.azure.com:4

config.endpoint

PRIMARY KEY

[redacted]

config.primaryKey

SECONDARY KEY

[redacted]

PRIMARY CONNECTION STRING

AccountEndpoint=https://contoso1.docu

SECONDARY CONNECTION STRING

AccountEndpoint=https://contoso1.docu

```
// 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 Azure Cosmos DB'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 an Azure Cosmos DB 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 an Azure Cosmos DB 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 in the app.js file to create 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.

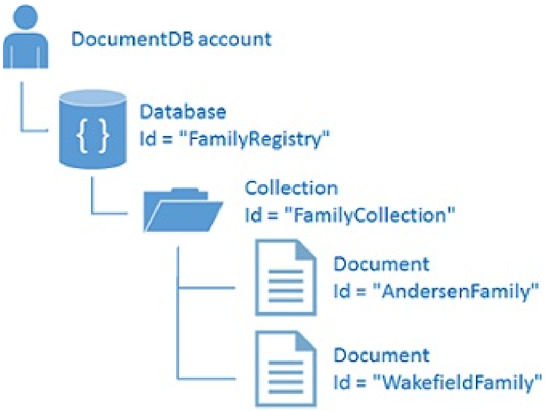
```
            } else {
                resolve(result);
            }
        });
    });
}

// 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.



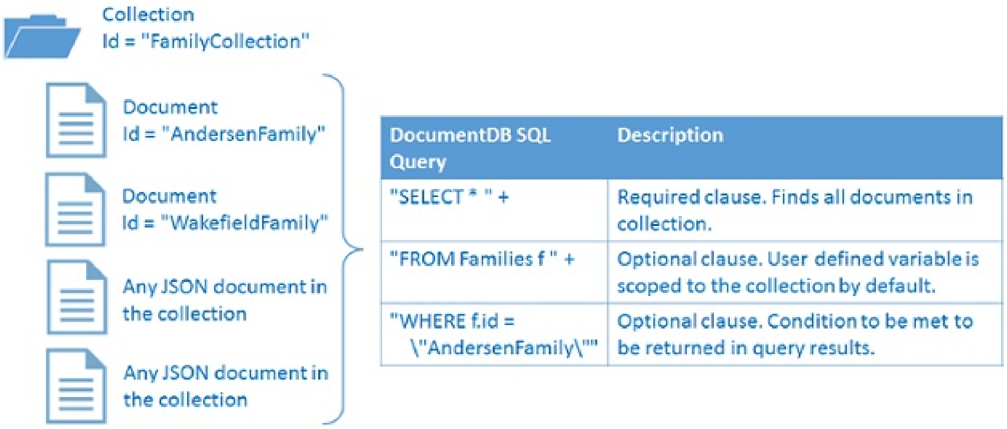
Step 8: Query Azure Cosmos DB resources

Azure Cosmos DB 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 in the app.js file. 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.



DocumentDB SQL Query	Description
"SELECT * " +	Required clause. Finds all documents in collection.
"FROM Families f " +	Optional clause. User defined variable is scoped to the collection by default.
"WHERE f.id = \"AndersenFamily\""	Optional clause. Condition to be met to be returned in query results.

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 Azure Cosmos DB documents.

Step 9: Replace a document

Azure Cosmos DB supports replacing JSON documents.

Copy and paste the **replaceFamilyDocument** function underneath the **queryCollection** function in the app.js file.

```
                }
                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 an Azure Cosmos DB document.

Step 10: Delete a document

Azure Cosmos DB supports deleting JSON documents.

Copy and paste the **deleteFamilyDocument** function underneath the **replaceFamilyDocument** 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 an Azure Cosmos DB 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 **cleanup** function underneath the **deleteFamilyDocument** function 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 **deleteFamilyDocument** 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 Azure Cosmos DB console application!

Get the complete Node.js tutorial solution

If you didn't have time to complete the steps in this tutorial, or just want to download the code, you can get it from GitHub.

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

- Azure Cosmos DB 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.

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

That's it, build it and you're on your way!

Next steps

- Want a more complex Node.js sample? See Build a Node.js web application using Azure Cosmos DB.
- Learn how to monitor an Azure Cosmos DB account.
- Run queries against our sample dataset in the Query Playground.
- Learn more about the programming model in the Develop section of the Azure Cosmos DB documentation

page.

Azure Cosmos DB: C++ console application tutorial for the DocumentDB API

5/30/2017 • 9 min to read • Edit Online

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

We'll cover:

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

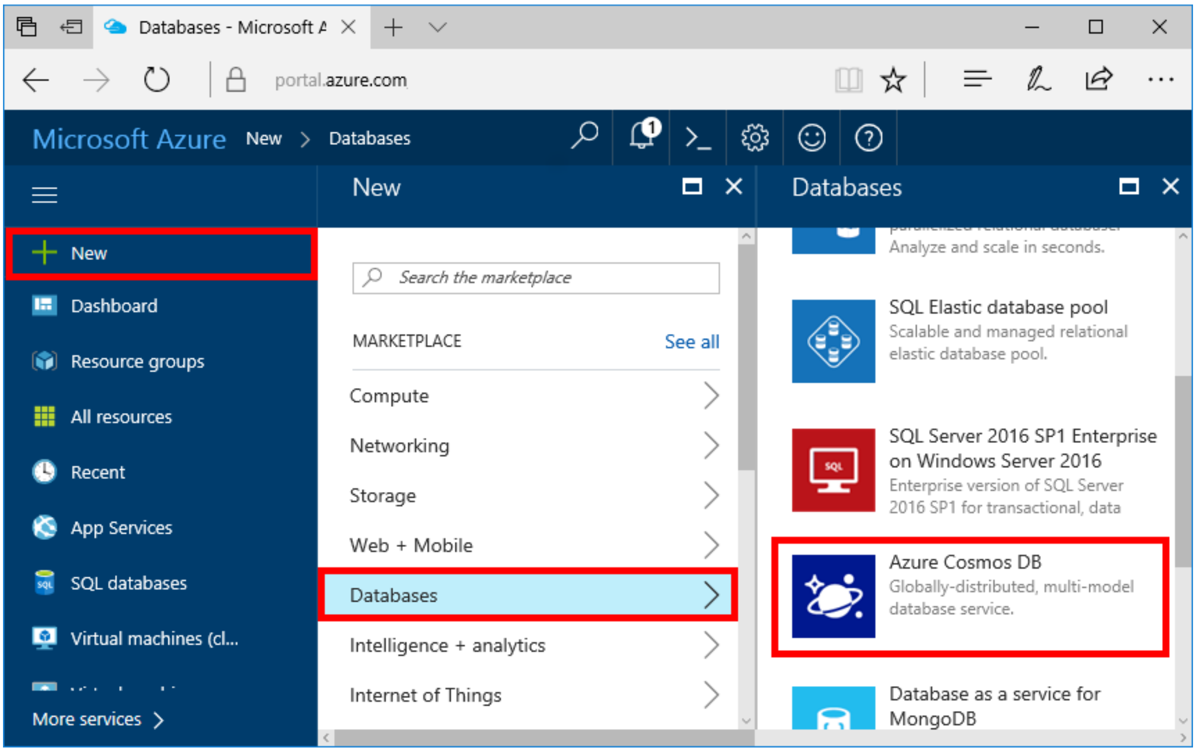
Please make sure you have the following:

- 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 an Azure Cosmos DB account

Let's create an Azure Cosmos DB account. If you already have an account you want to use, you can skip ahead to Setup your C++ application.

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

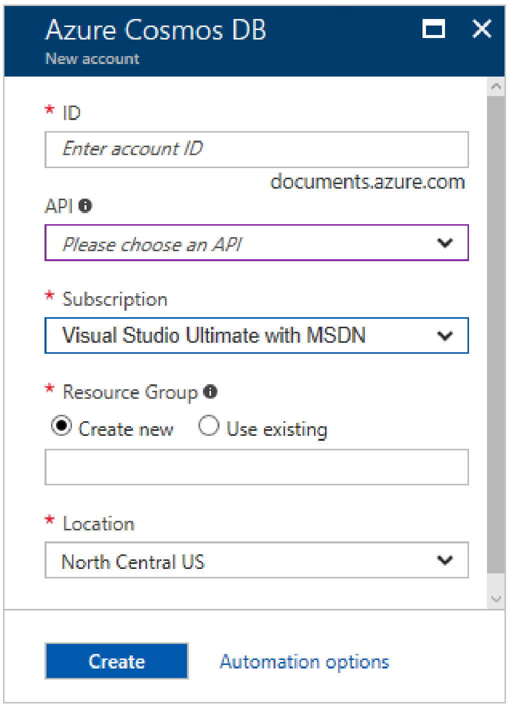


3. On the **New account** blade, specify the configuration that you want for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article we program against the DocumentDB API, so choose **SQL (DocumentDB)** as you fill out the form. But if you have graph data for a social media app, or key/value (table) data, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

   Complete the fields on the **New account** blade, using the information in the following screenshot as a guide. When you set up your account, choose unique values that do not match those in the screenshot.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

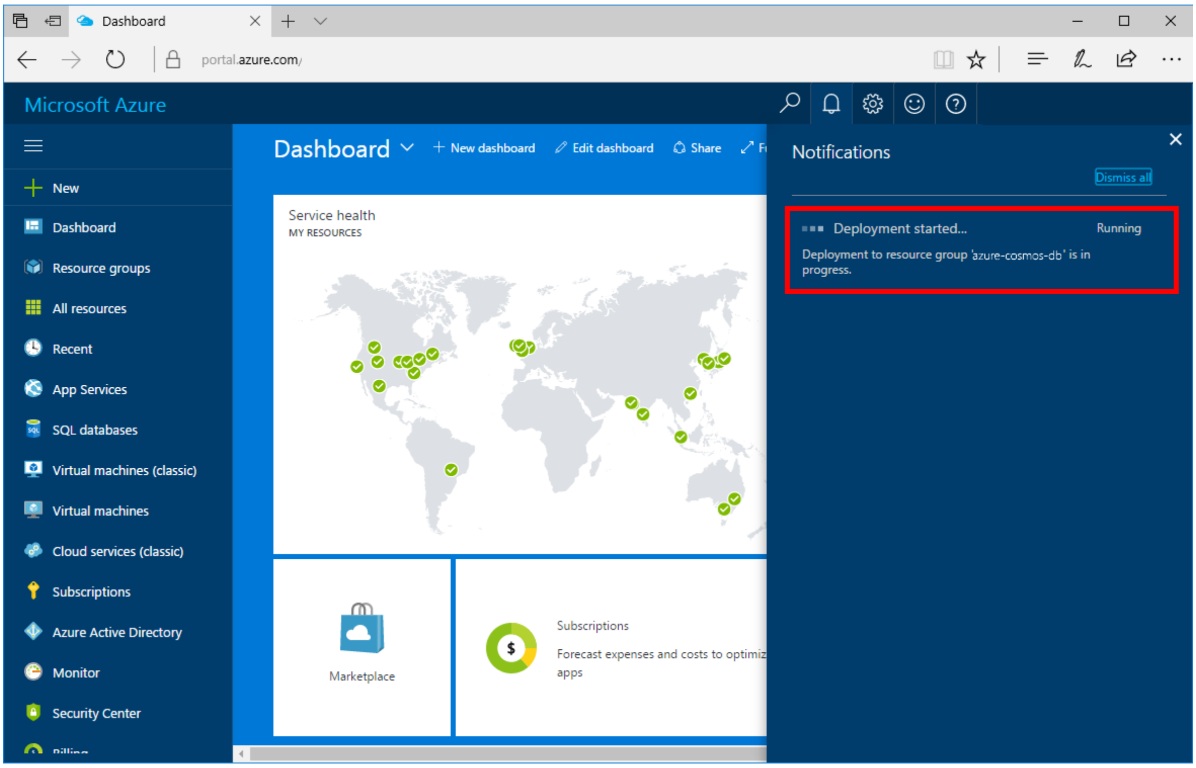
* Subscription
Visual Studio Ultimate with MSDN

* Resource Group
● Create new ○ Use existing

* Location
North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that identifies your Azure Cosmos DB account. The string *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 through 50 characters.
API	SQL (DocumentDB)	We program against the DocumentDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for your Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource-group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location that's closest to your users to give them the fastest access to the data.

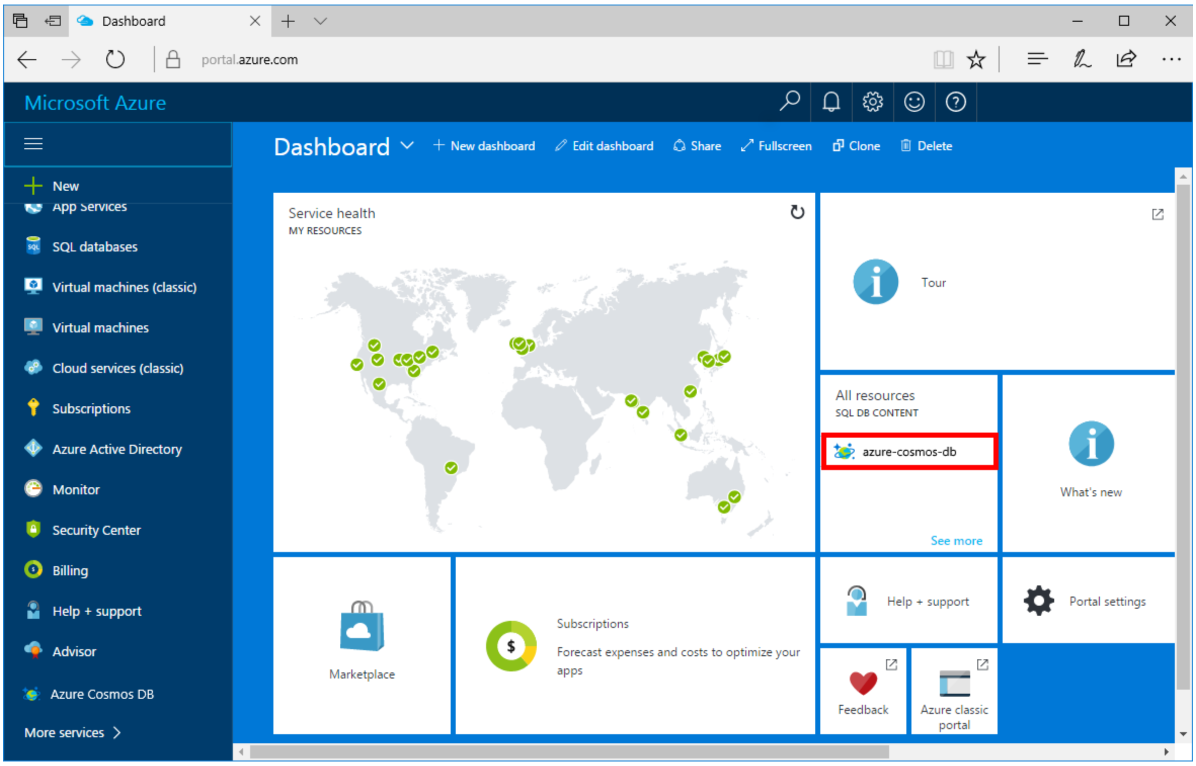
4. Click **Create** to create the account.
5. On the top toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

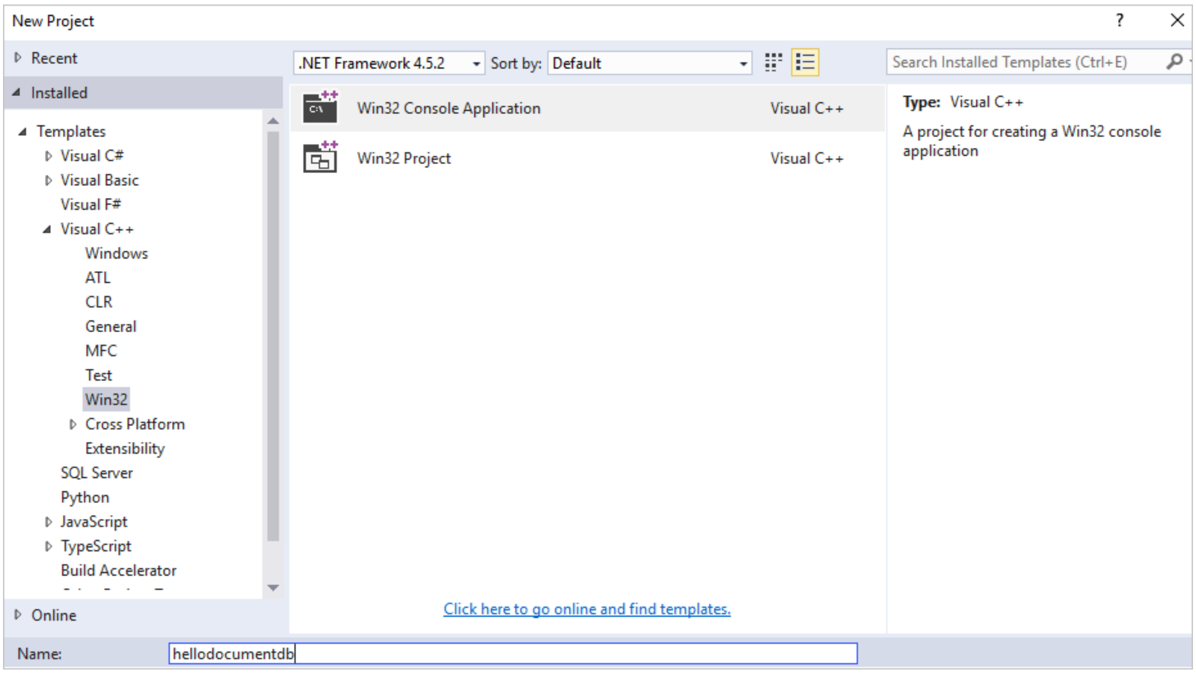
Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the **All Resources** tile.



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**.



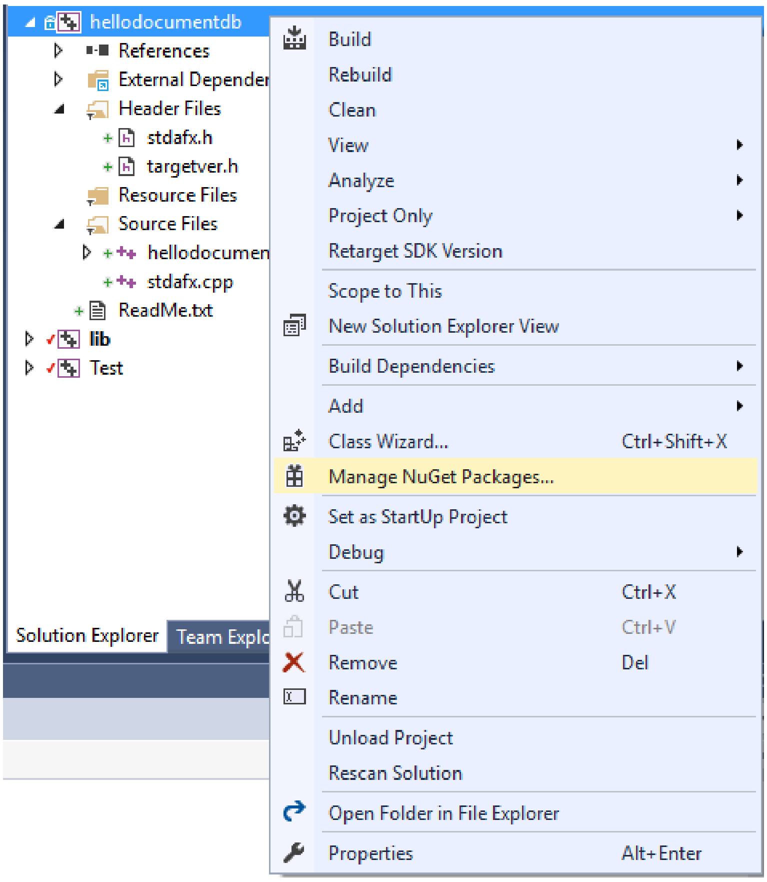
.NET Framework 4.5.2 Sort by: Default

Type: Visual C++

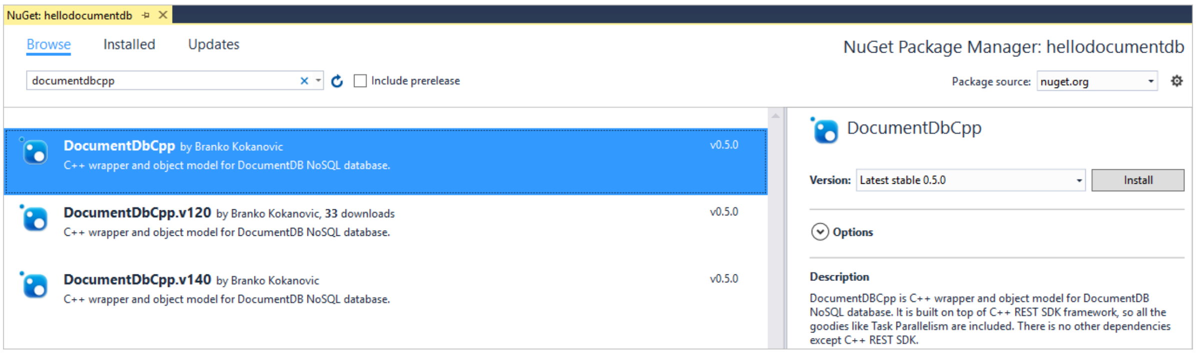
A project for creating a Win32 console application

Name: hellodocumentdb

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.



documentdbcpp

Package source: nuget.org

DocumentDbCpp by Branko Kokanovic v0.5.0
C++ wrapper and object model for DocumentDB NoSQL database.

DocumentDbCpp.v120 by Branko Kokanovic, 33 downloads v0.5.0
C++ wrapper and object model for DocumentDB NoSQL database.

DocumentDbCpp.v140 by Branko Kokanovic v0.5.0
C++ wrapper and object model for DocumentDB NoSQL database.

DocumentDbCpp

Version: Latest stable 0.5.0

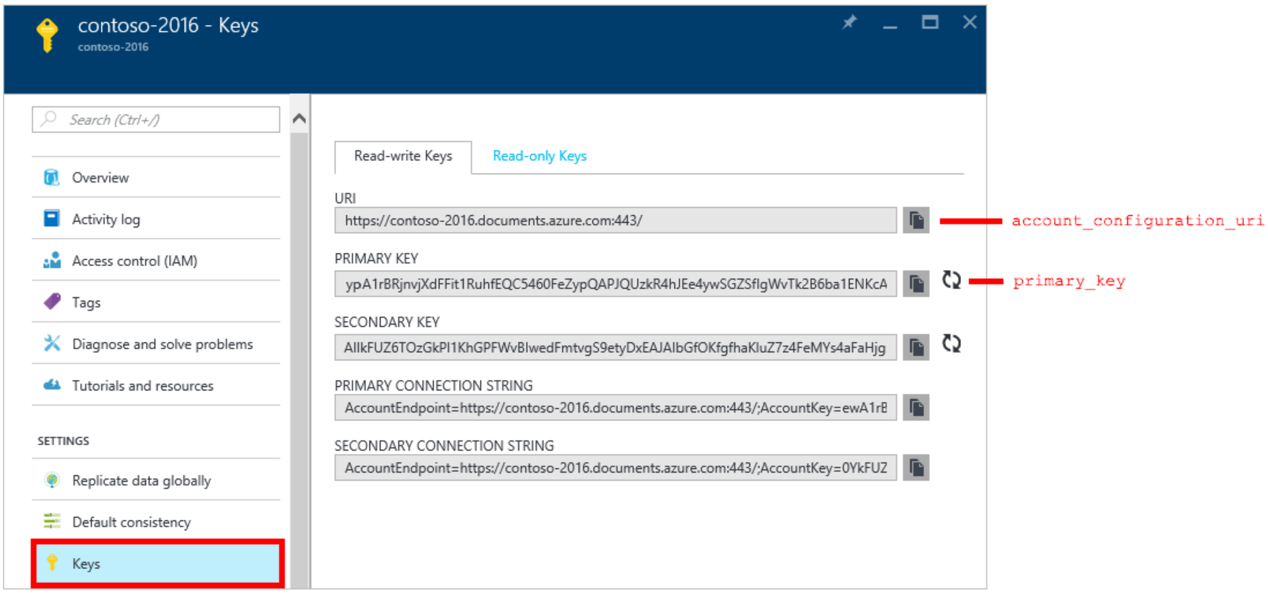
Description

DocumentDBCpp is C++ wrapper and object model for DocumentDB NoSQL database. It is built on top of C++ REST SDK framework, so all the goodies like Task Parallelism are included. There is no other dependencies except C++ REST SDK.

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 Azure Cosmos DB database

Bring up Azure portal and traverse to the Azure Cosmos DB 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.



URI
https://contoso-2016.documents.azure.com:443/

PRIMARY KEY
ypA1rBRjnvjXdFFit1RuhfEQC5460FeZypQAPJQUzkR4hJEe4ywSGZSflgWvTk2B6ba1ENKcA

SECONDARY KEY
AIlkFUZ6TOzGkPI1KhGPFWvBlwedFmtvgS9etyDxEAJAIbGfOKfgfhaKluZ7z4FeMYs4aFaHjg

PRIMARY CONNECTION STRING
AccountEndpoint=https://contoso-2016.documents.azure.com:443/;AccountKey=ewA1rB

SECONDARY CONNECTION STRING
AccountEndpoint=https://contoso-2016.documents.azure.com:443/;AccountKey=0YkFUZ

Step 4: Connect to an Azure Cosmos DB account

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

```
#include <cpprest/json.h>
#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 Azure Cosmos DB 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 Azure Cosmos DB 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 Azure Cosmos DB. 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 Azure Cosmos DB hierarchical resource model and concepts in Azure Cosmos DB 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.

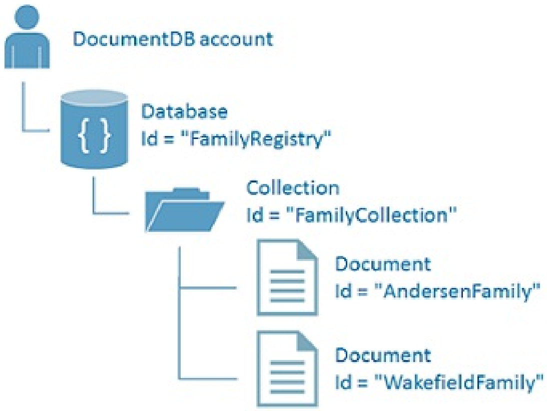
```
try {
  shared_ptr<Database> db = client.CreateDatabase(L"FamilyRegistry");
  shared_ptr<Collection> coll = db->CreateCollection(L"FamilyCollection");
} catch (DocumentDBRuntimeException ex) {
  wcout << ex.message();
}
```

Step 6: Create a document

Documents are user-defined (arbitrary) JSON content. You can now insert a document into Azure Cosmos DB. 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 an Azure Cosmos DB database, collection, and documents, which you can query in Document Explorer in Azure portal.



Step 7: Query Azure Cosmos DB resources

Azure Cosmos DB supports rich queries against JSON documents stored in each collection. The following sample code shows a query made using 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 (DocumentDBRuntimeException ex) {
    wcout << ex.message();
  }
}
```

Step 8: Replace a document

Azure Cosmos DB 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 (DocumentDBRuntimeException ex) {
    throw;
  }
}
```

Step 9: Delete a document

Azure Cosmos DB 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 (DocumentDBRuntimeException 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 (DocumentDBRuntimeException ex) {
    wcout << ex.message();
  }
}
```

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

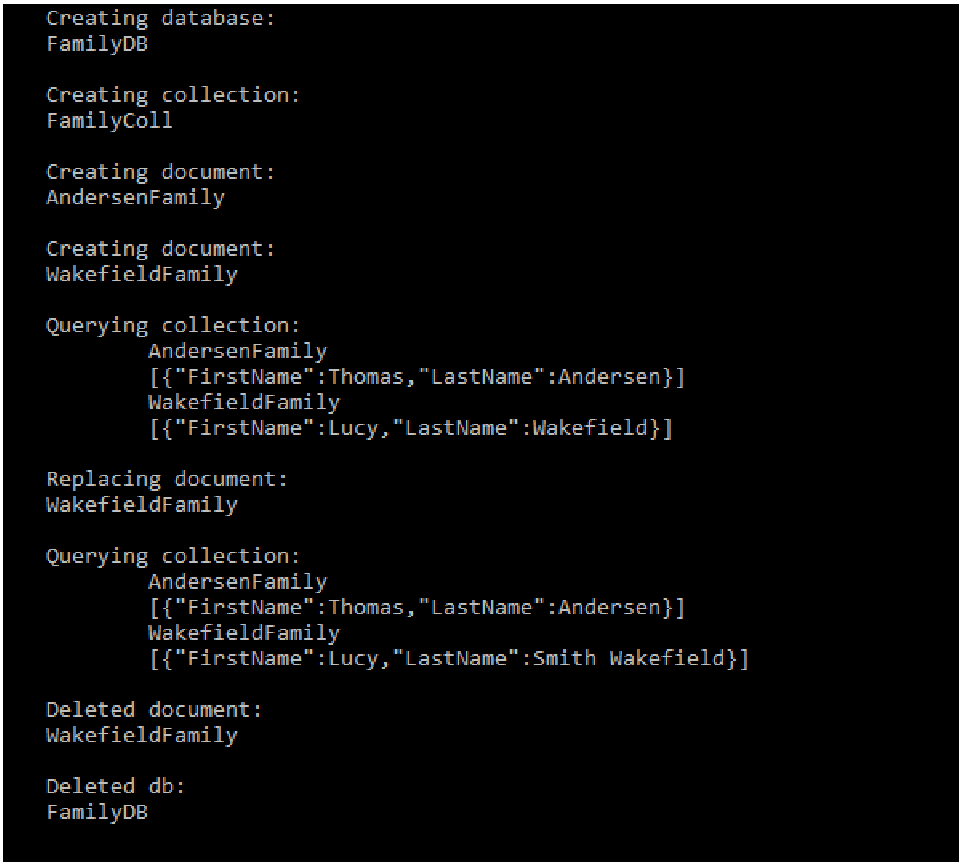
We have now added code to create, query, modify, and delete different Azure Cosmos DB resources. Let us now wire this up by adding calls to these different functions from our main function in hellodocumentdb.cpp along with some diagnostic messages.

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 (DocumentDBRuntimeException 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 Azure Cosmos DB 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:

- Azure Cosmos DB account.
- The GetStarted solution available on GitHub.

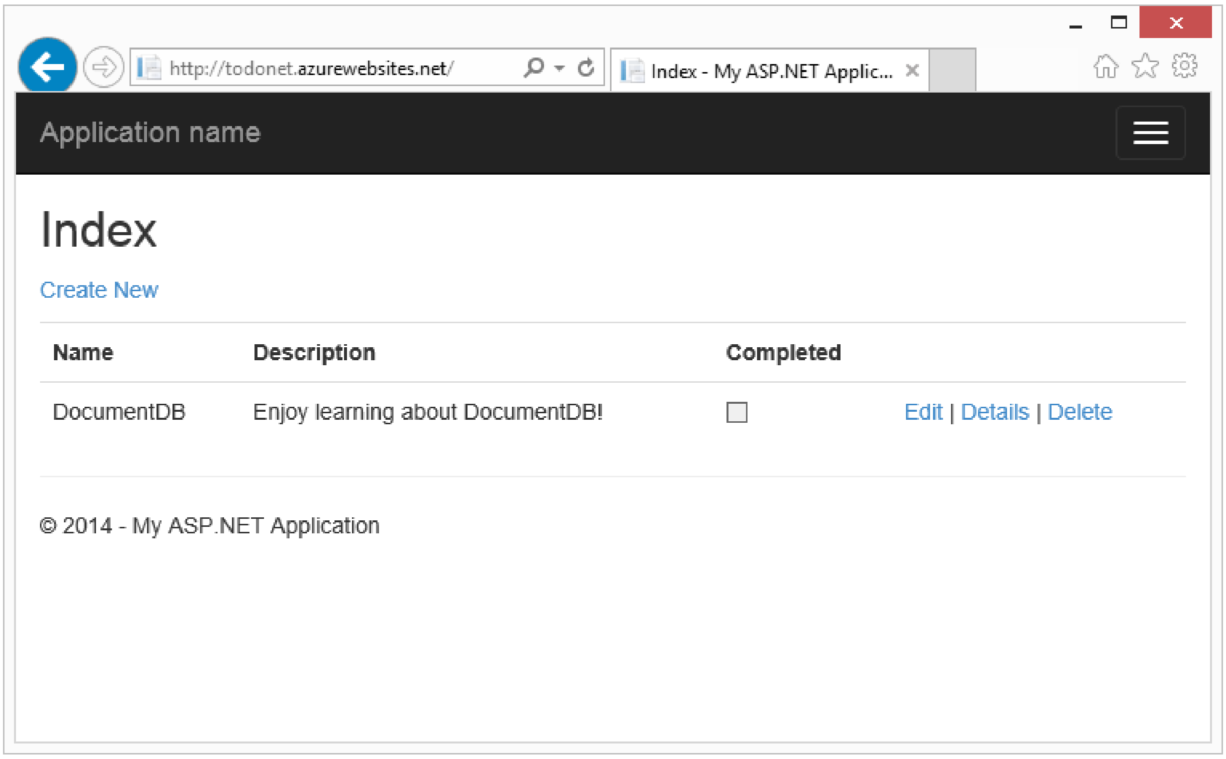
Next steps

- Learn how to monitor an Azure Cosmos DB account.
- Run queries against our sample dataset in the Query Playground.
- Learn more about the programming model in the Develop section of the Azure Cosmos DB documentation page.

ASP.NET MVC Tutorial: Web application development with Azure Cosmos DB

5/30/2017 • 20 min to read • Edit Online

To highlight how you can efficiently leverage Azure Cosmos DB 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 Cosmos DB. The tasks will be stored as JSON documents in Azure Cosmos DB.



http://todonet.azurewebsites.net/
Index - My ASP.NET Applic...
Application name

Index

Create New

Name	Description	Completed	
DocumentDB	Enjoy learning about DocumentDB!	□	Edit | Details | Delete

© 2014 - My ASP.NET Application

This walk-through shows you how to use the Azure Cosmos DB 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 Azure Cosmos DB, and not the ASP.NET MVC components, see Build an Azure Cosmos DB 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 Cosmos DB 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.

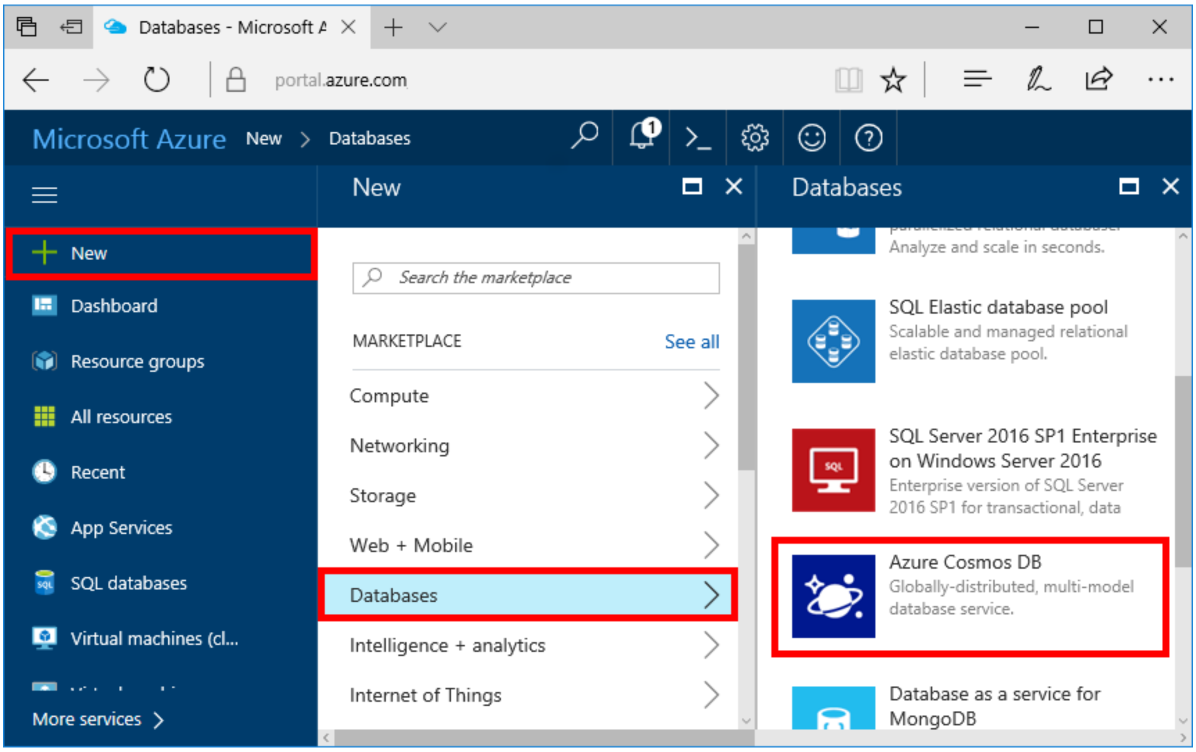
- Azure SDK for .NET version 2.5.1 or higher, available through the Microsoft Web Platform Installer.

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 an Azure Cosmos DB database account

Let's start by creating an Azure Cosmos DB account. If you already have an account or if you are using the Azure Cosmos DB Emulator for this tutorial, you can skip to Create a new ASP.NET MVC application.

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

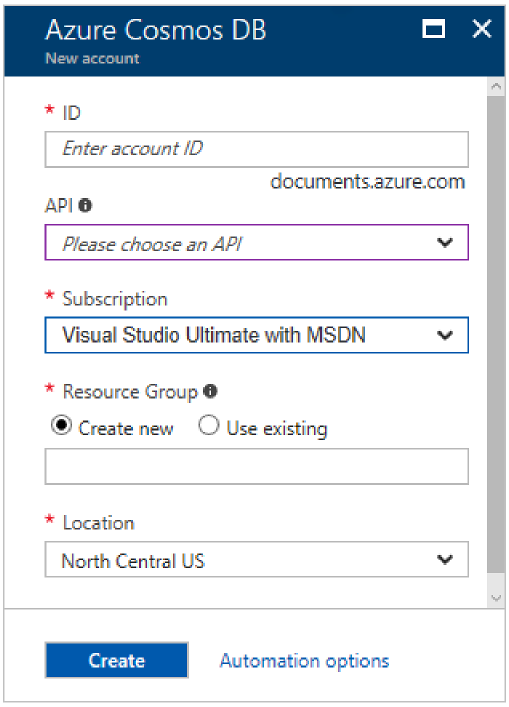


3. On the **New account** blade, specify the configuration that you want for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article we program against the DocumentDB API, so choose **SQL (DocumentDB)** as you fill out the form. But if you have graph data for a social media app, or key/value (table) data, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

   Complete the fields on the **New account** blade, using the information in the following screenshot as a guide. When you set up your account, choose unique values that do not match those in the screenshot.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

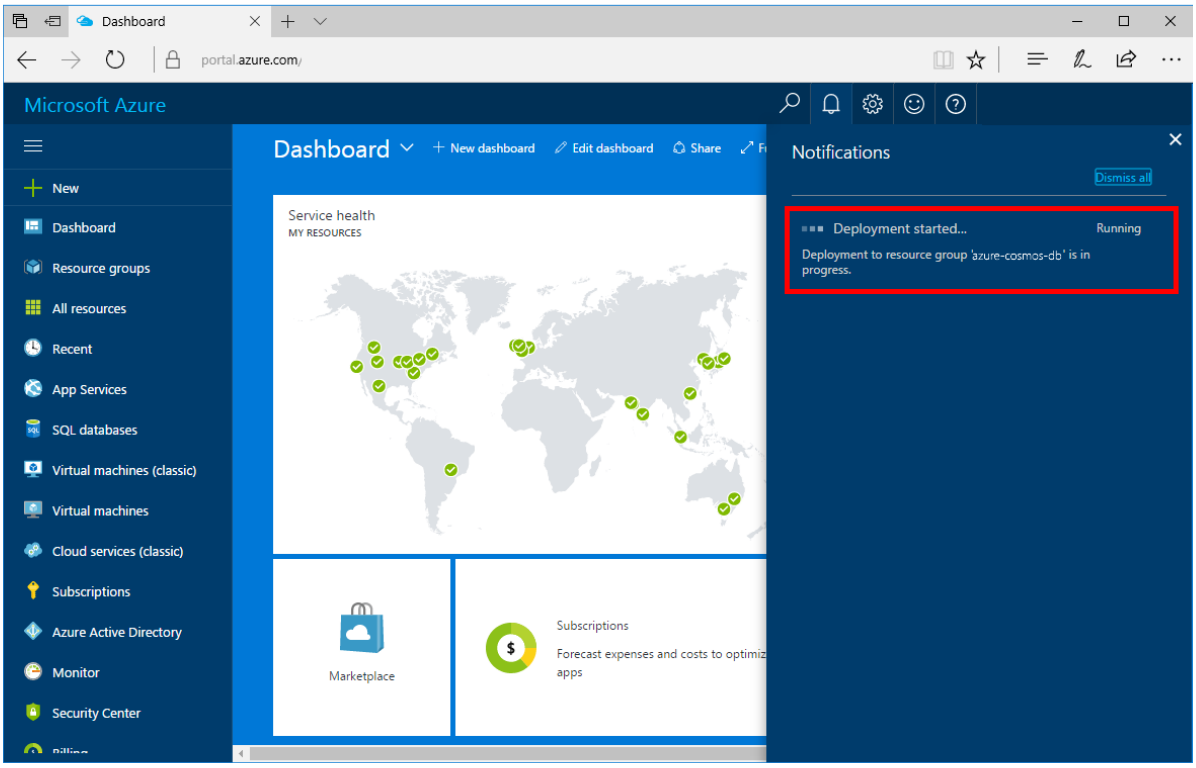
* Subscription
Visual Studio Ultimate with MSDN

* Resource Group
● Create new ○ Use existing

* Location
North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that identifies your Azure Cosmos DB account. The string *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 through 50 characters.
API	SQL (DocumentDB)	We program against the DocumentDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for your Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource-group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location that's closest to your users to give them the fastest access to the data.

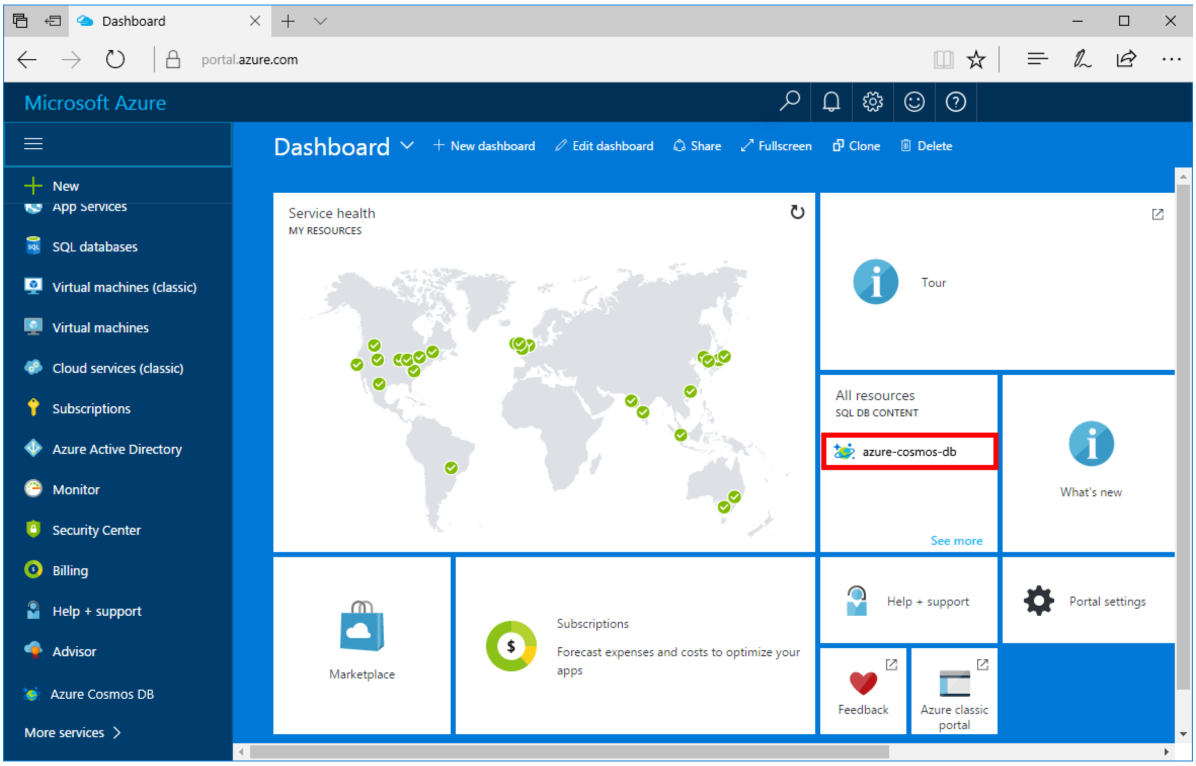
4. Click **Create** to create the account.
5. On the top toolbar, click **Notifications** to monitor the deployment process.



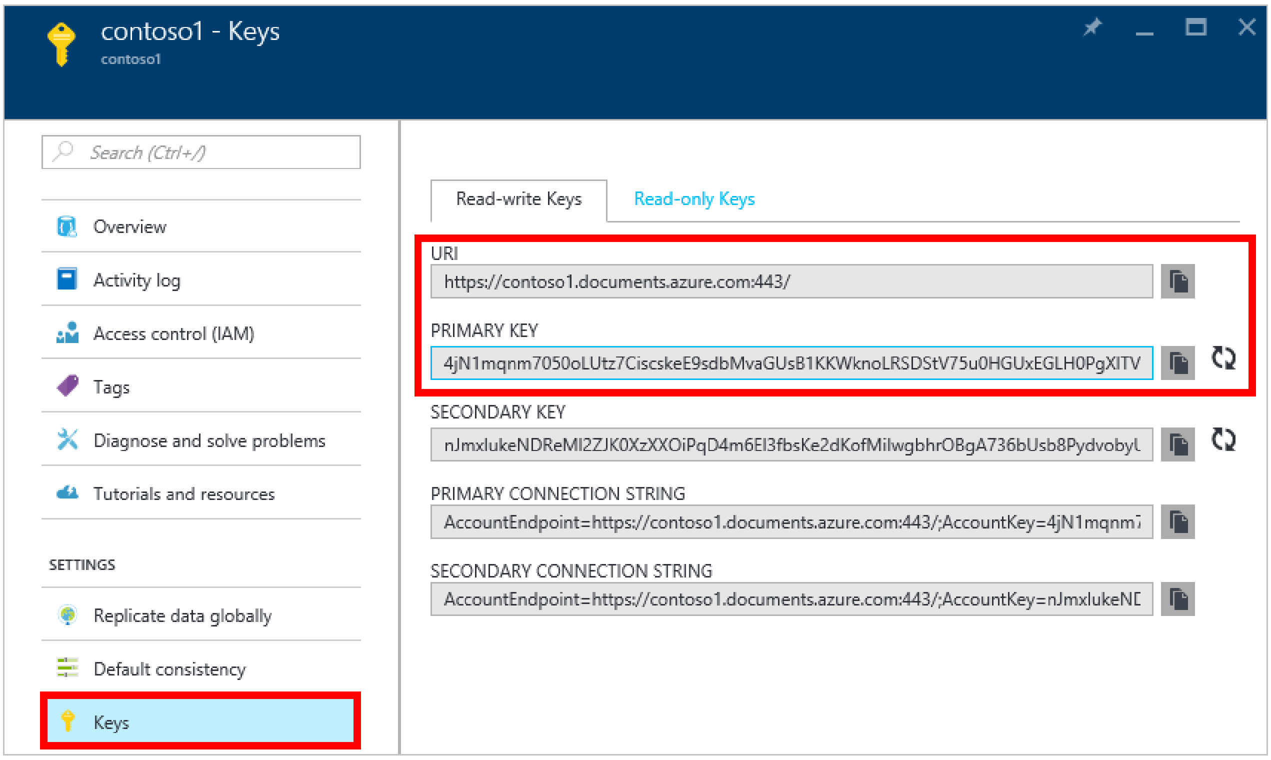
Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the **All Resources** tile.



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



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=nJmxlukeND

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

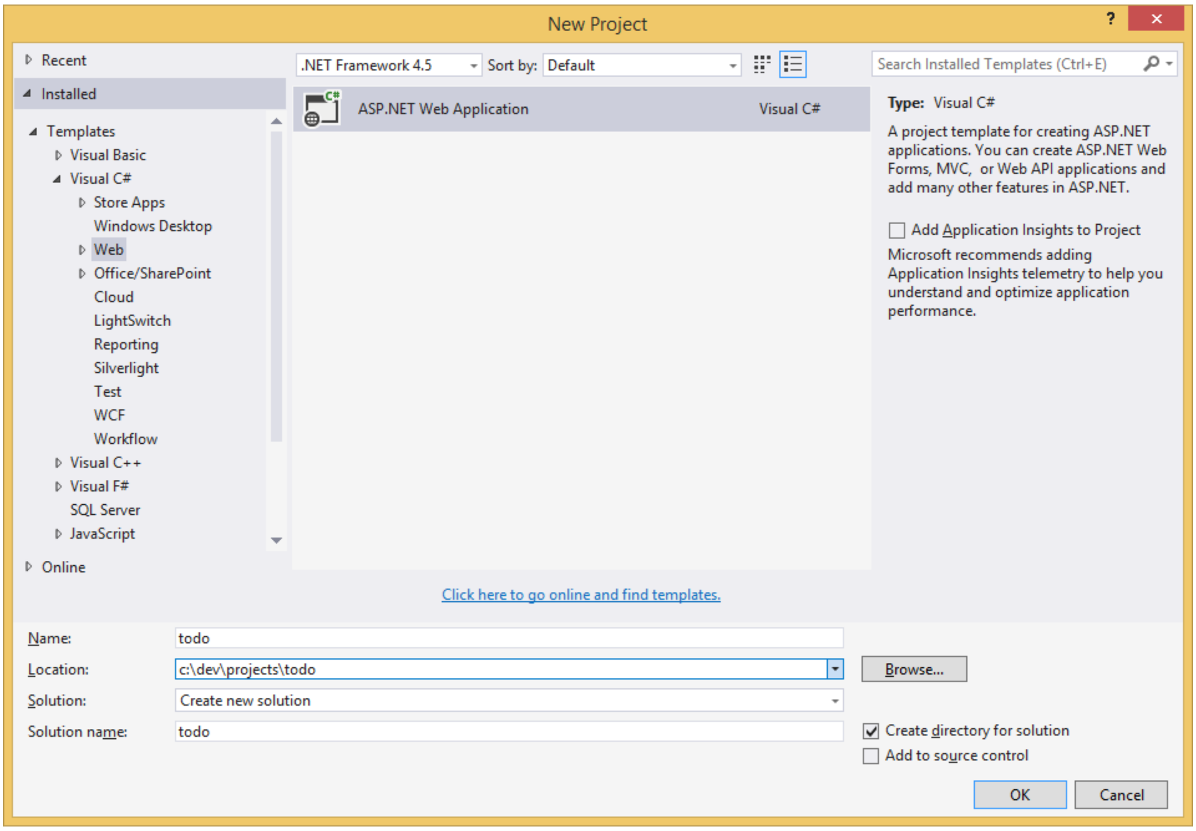
Step 2: Create a new ASP.NET MVC application

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**.



.NET Framework 4.5 Sort by: Default

Type: Visual C#

A project template for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.

□ Add Application Insights to Project

Microsoft recommends adding Application Insights telemetry to help you understand and optimize application performance.

Name: todo

Location: c:\dev\projects\todo

Solution: Create new solution

Solution name: todo

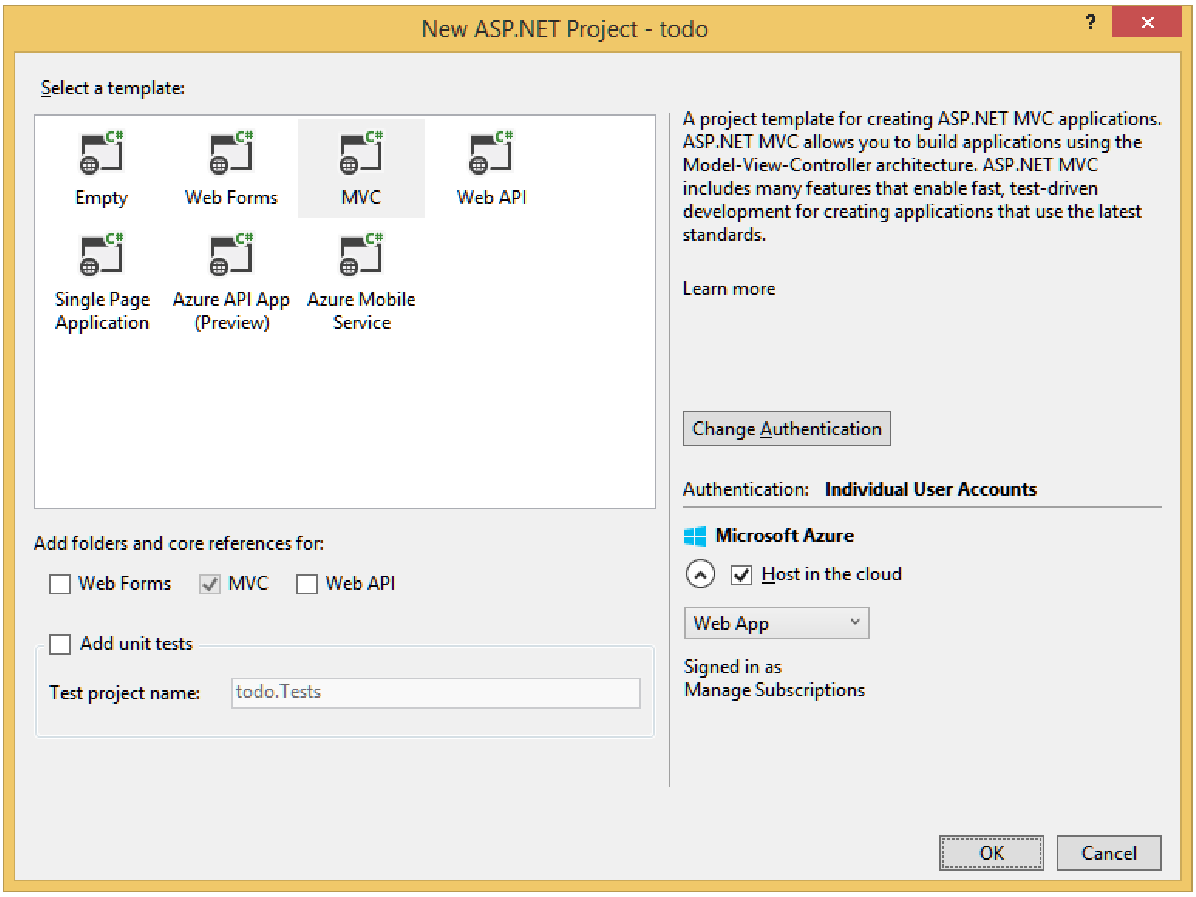
☑ Create directory for solution

□ Add to source control

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.



A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features that enable fast, test-driven development for creating applications that use the latest standards.

Authentication: **Individual User Accounts**

Add folders and core references for:

□ Web Forms ☑ MVC □ Web API

□ Add unit tests

Test project name: todo.Tests

☑ Host in the cloud

Web App

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

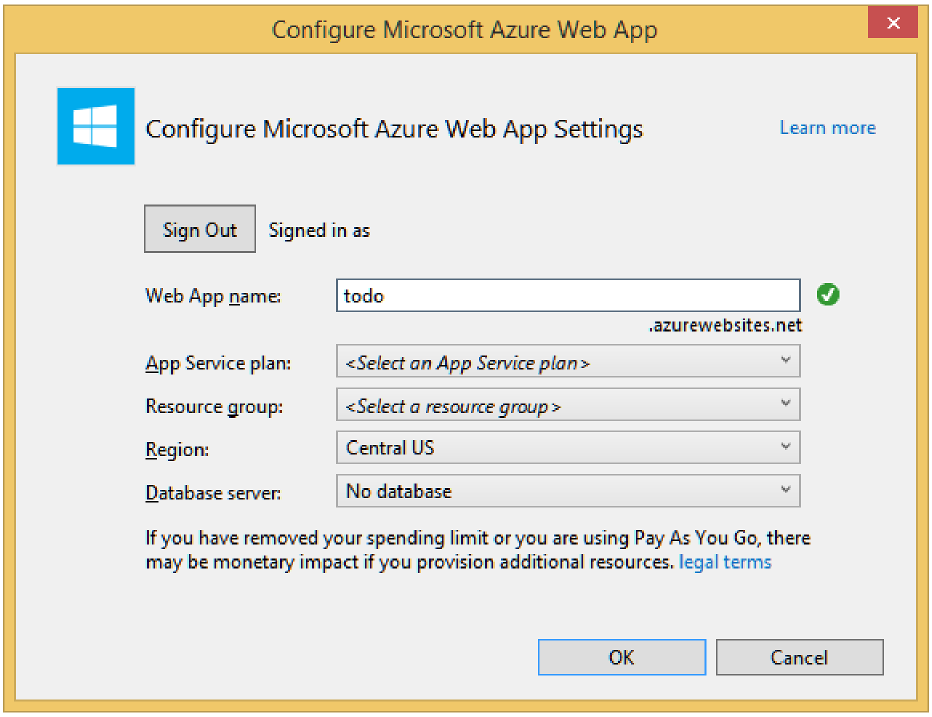
6. If you plan on hosting your application in Azure then select **Host in the cloud** on the lower right to have Azure host the application. We've selected to host in the cloud, and to run the application hosted in an Azure Website. Selecting this option will preprovision an Azure Website for you and 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 Cosmos DB 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.



Configure Microsoft Azure Web App

Configure Microsoft Azure Web App Settings

Signed in as

Web App name: todo .azurewebsites.net

App Service plan: <Select an App Service plan>

Resource group: <Select a resource group>

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

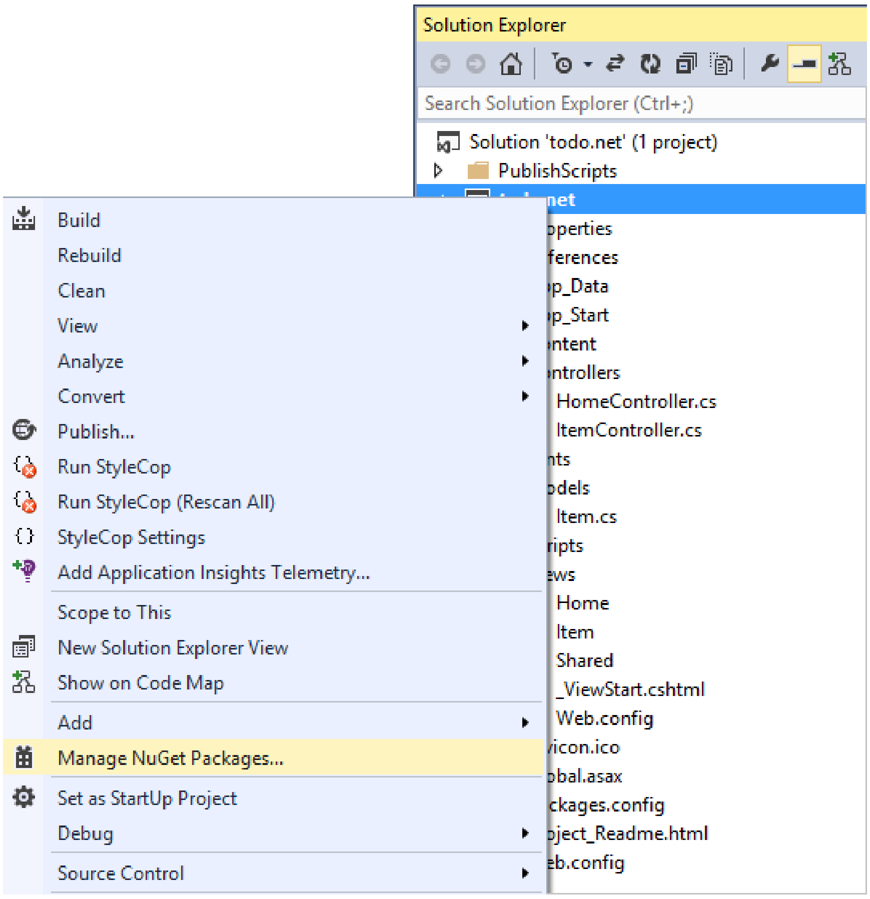
9. Once Visual Studio has finished creating the boilerplate MVC application you have an empty ASP.NET application that you can run locally.

   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 Azure Cosmos DB to this project and building our application.

Step 3: Add Azure Cosmos DB 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 Cosmos DB to our MVC web application.

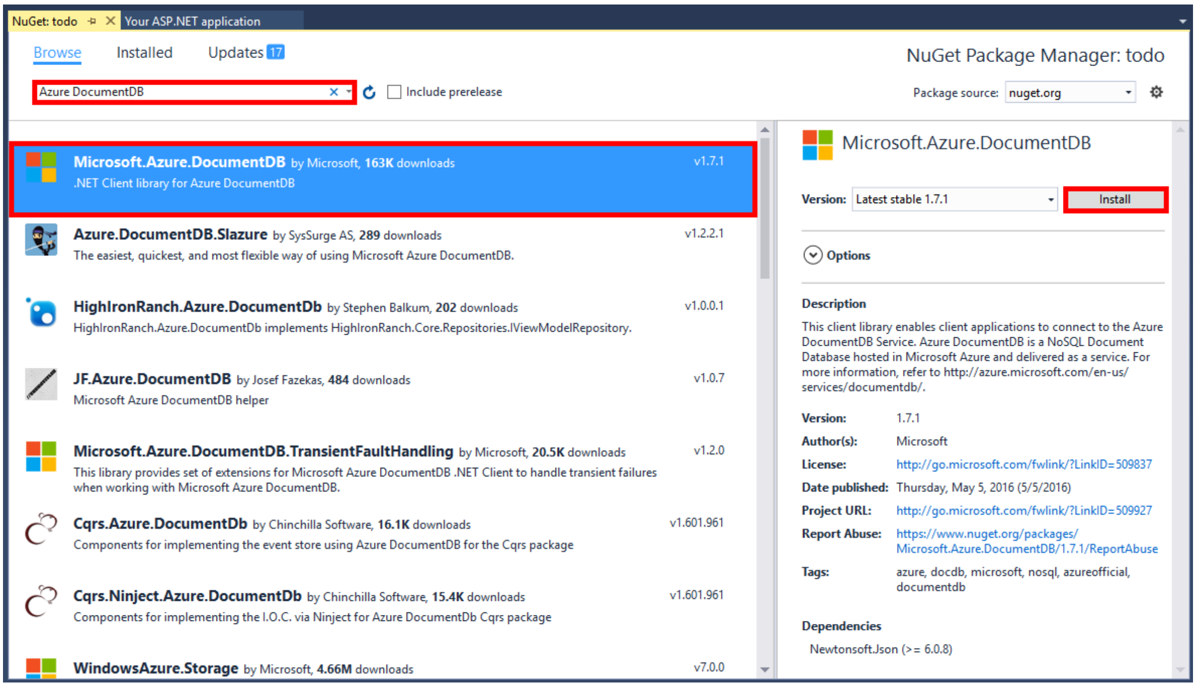
1. The DocumentDB .NET SDK is packaged and distributed as a NuGet package. To get the NuGet package in Visual Studio, use the NuGet package manager in Visual Studio by right-clicking on the project in **Solution Explorer** and then clicking **Manage NuGet Packages**.



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

2. In the NuGet **Browse** box, type ***Azure Cosmos DB***.

   From the results, install the **Microsoft Azure Cosmos DB Client Library** package. This will download and install the Azure Cosmos DB 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.



Azure DocumentDB

Package source: nuget.org

Microsoft.Azure.DocumentDB by Microsoft, 163K downloads v1.7.1
.NET Client library for Azure DocumentDB

Azure.DocumentDB.Slazure by SysSurge AS, 289 downloads v1.2.2.1
The easiest, quickest, and most flexible way of using Microsoft Azure DocumentDB.

HighIronRanch.Azure.DocumentDb by Stephen Balkum, 202 downloads v1.0.0.1
HighIronRanch.Azure.DocumentDb implements HighIronRanch.Core.Repositories.IViewModelRepository.

JF.Azure.DocumentDB by Josef Fazekas, 484 downloads v1.0.7
Microsoft Azure DocumentDB helper

Microsoft.Azure.DocumentDB.TransientFaultHandling by Microsoft, 20.5K downloads v1.2.0
This library provides set of extensions for Microsoft Azure DocumentDB .NET Client to handle transient failures when working with Microsoft Azure DocumentDB.

Cqrs.Azure.DocumentDb by Chinchilla Software, 16.1K downloads v1.601.961
Components for implementing the event store using Azure DocumentDB for the Cqrs package

Cqrs.Ninject.Azure.DocumentDb by Chinchilla Software, 15.4K downloads v1.601.961
Components for implementing the I.O.C. via Ninject for Azure DocumentDb Cqrs package

WindowsAzure.Storage by Microsoft, 4.66M downloads v7.0.0

Microsoft.Azure.DocumentDB

Version: Latest stable 1.7.1

Description
This client library enables client applications to connect to the Azure DocumentDB Service. Azure DocumentDB is a NoSQL Document Database hosted in Microsoft Azure and delivered as a service. For more information, refer to http://azure.microsoft.com/en-us/services/documentdb/.

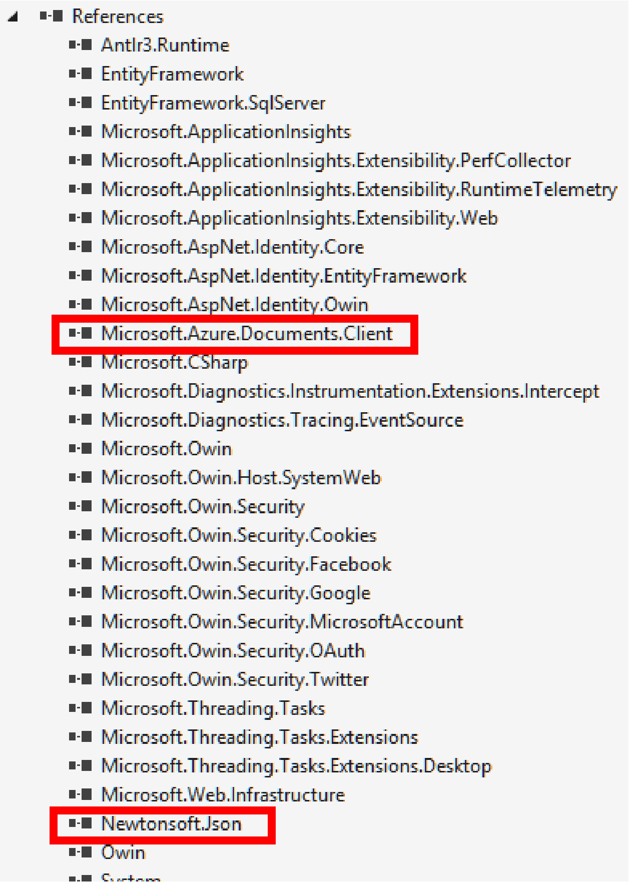
Version: 1.7.1
Author(s): Microsoft
License: http://go.microsoft.com/fwlink/?LinkID=509837
Date published: Thursday, May 5, 2016 (5/5/2016)
Project URL: http://go.microsoft.com/fwlink/?LinkID=509927
Report Abuse: https://www.nuget.org/packages/Microsoft.Azure.DocumentDB/1.7.1/ReportAbuse
Tags: azure, docdb, microsoft, nosql, azureofficial, documentdb

Dependencies
Newtonsoft.Json (>= 6.0.8)

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
```

3. 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.



References
- Antlr3.Runtime
- EntityFramework
- EntityFramework.SqlServer
- Microsoft.ApplicationInsights
- Microsoft.ApplicationInsights.Extensibility.PerfCollector
- Microsoft.ApplicationInsights.Extensibility.RuntimeTelemetry
- Microsoft.ApplicationInsights.Extensibility.Web
- Microsoft.AspNet.Identity.Core
- Microsoft.AspNet.Identity.EntityFramework
- Microsoft.AspNet.Identity.Owin
- Microsoft.Azure.Documents.Client
- Microsoft.CSharp
- Microsoft.Diagnostics.Instrumentation.Extensions.Intercept
- Microsoft.Diagnostics.Tracing.EventSource
- Microsoft.Owin
- Microsoft.Owin.Host.SystemWeb
- Microsoft.Owin.Security
- Microsoft.Owin.Security.Cookies
- Microsoft.Owin.Security.Facebook
- Microsoft.Owin.Security.Google
- Microsoft.Owin.Security.MicrosoftAccount
- Microsoft.Owin.Security.OAuth
- Microsoft.Owin.Security.Twitter
- Microsoft.Threading.Tasks
- Microsoft.Threading.Tasks.Extensions
- Microsoft.Threading.Tasks.Extensions.Desktop
- Microsoft.Web.Infrastructure
- Newtonsoft.Json
- Owin
- System

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.

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

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

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

```
using Newtonsoft.Json;
```

4. 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 Azure Cosmos DB 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.

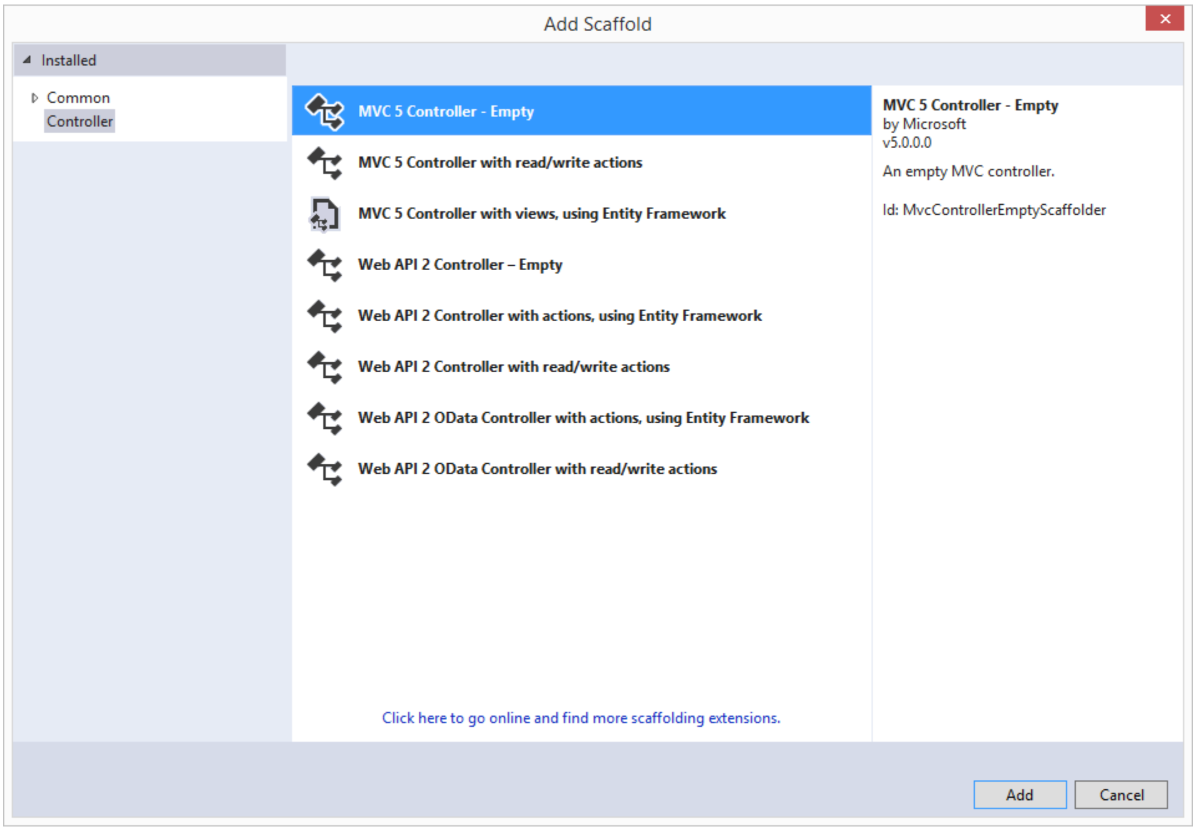
Add a controller

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**.

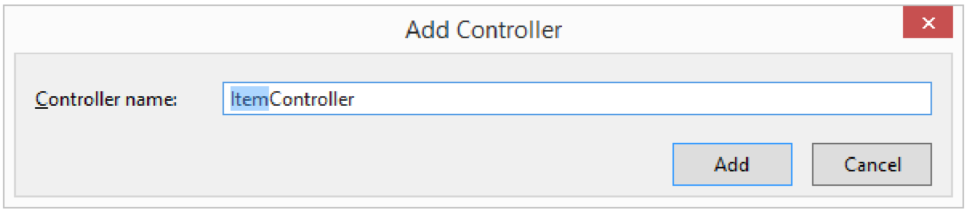

Add Scaffold
Installed
Common
Controller
MVC 5 Controller - Empty
MVC 5 Controller with read/write actions
MVC 5 Controller with views, using Entity Framework
Web API 2 Controller – Empty
Web API 2 Controller with actions, using Entity Framework
Web API 2 Controller with read/write actions
Web API 2 OData Controller with actions, using Entity Framework
Web API 2 OData Controller with read/write actions
Click here to go online and find more scaffolding extensions.
Add
Cancel

MVC 5 Controller - Empty
by Microsoft
v5.0.0.0

An empty MVC controller.

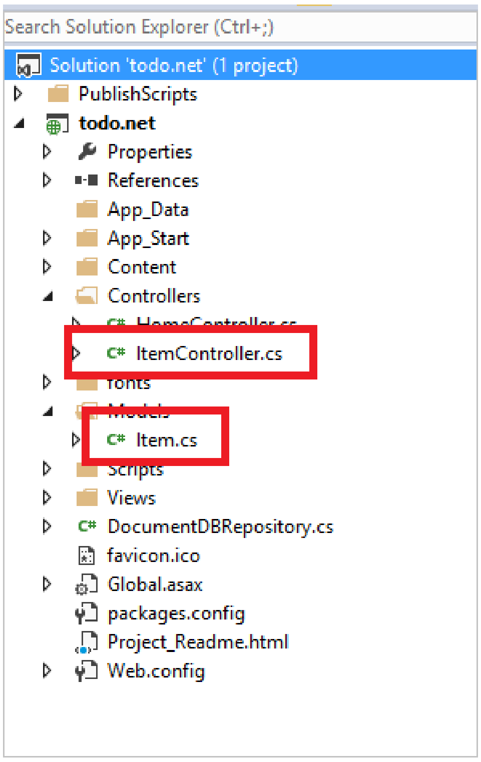
Id: MvcControllerEmptyScaffolder

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



Controller name: 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.

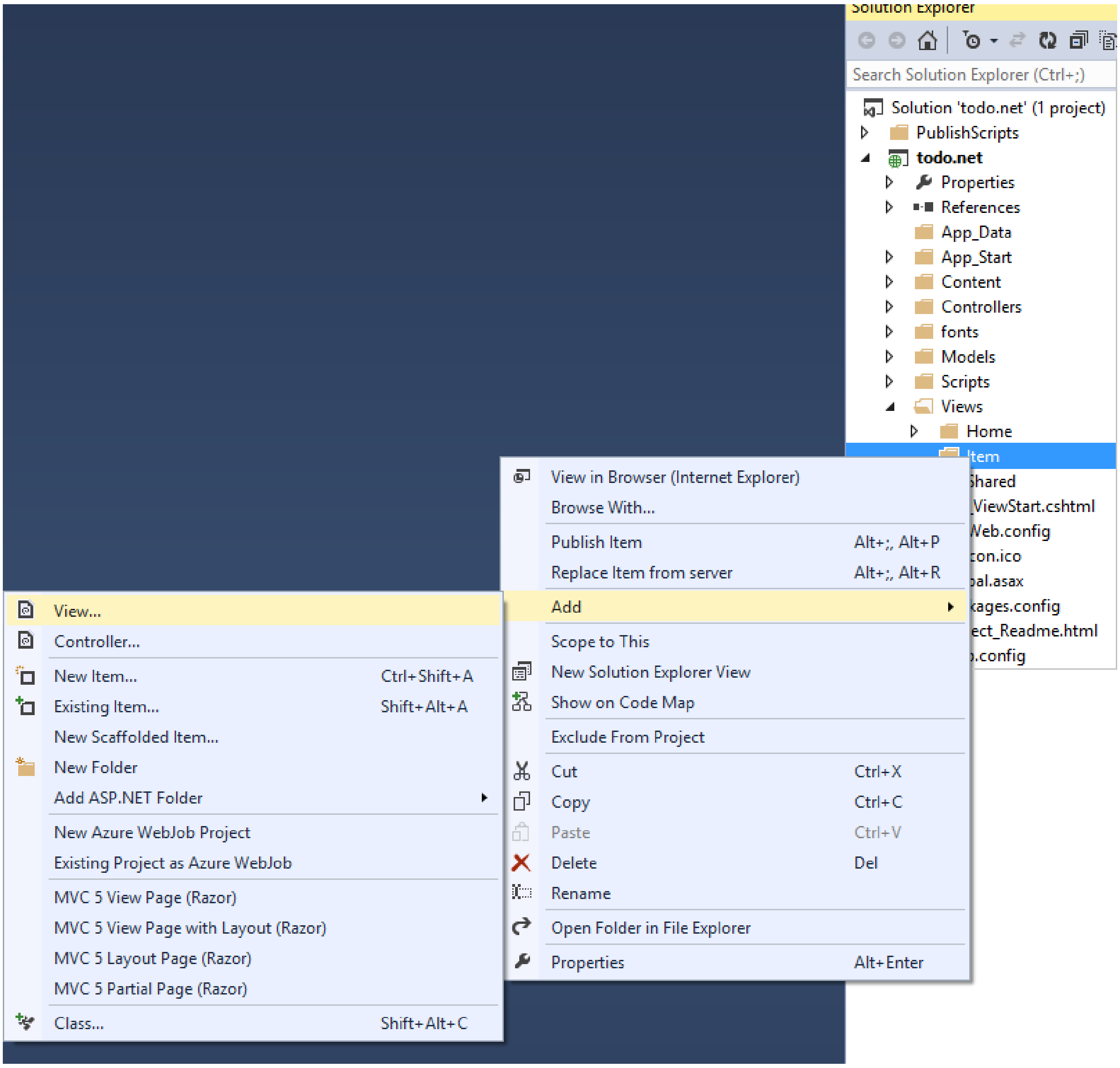
Add views

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.

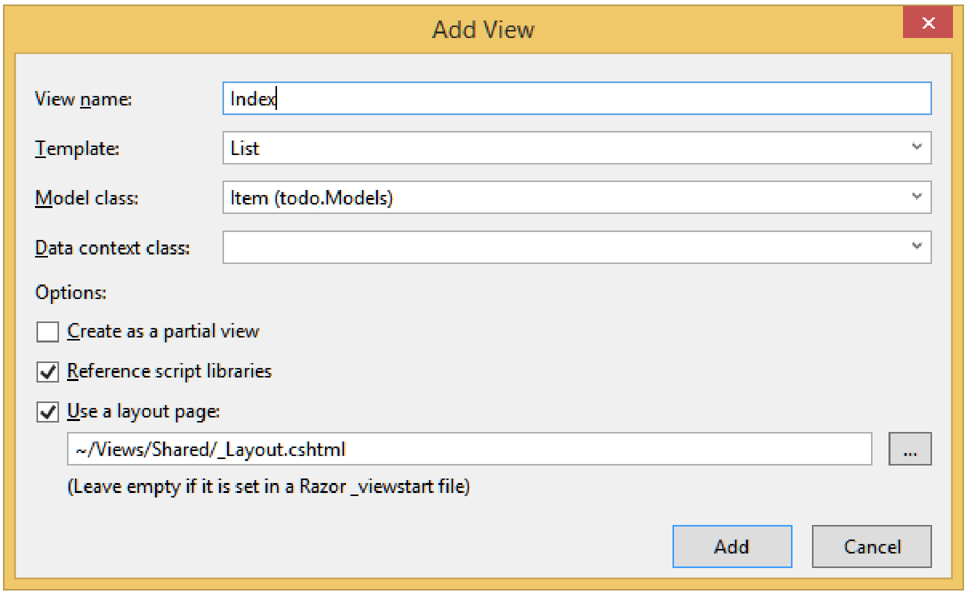
Add an Item Index view

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***.



View name: Index

Template: List

Model class: Item (todo.Models)

Data context class:

Options:

☐ Create as a partial view

☑ Reference script libraries

☑ Use a layout page:

~/Views/Shared/_Layout.cshtml

(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 Azure Cosmos DB

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

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 Azure Cosmos DB. 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 Azure Cosmos DB offer types please refer to Azure Cosmos DB 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"/>
```

4. 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.

5. 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;
}
```

6. 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.

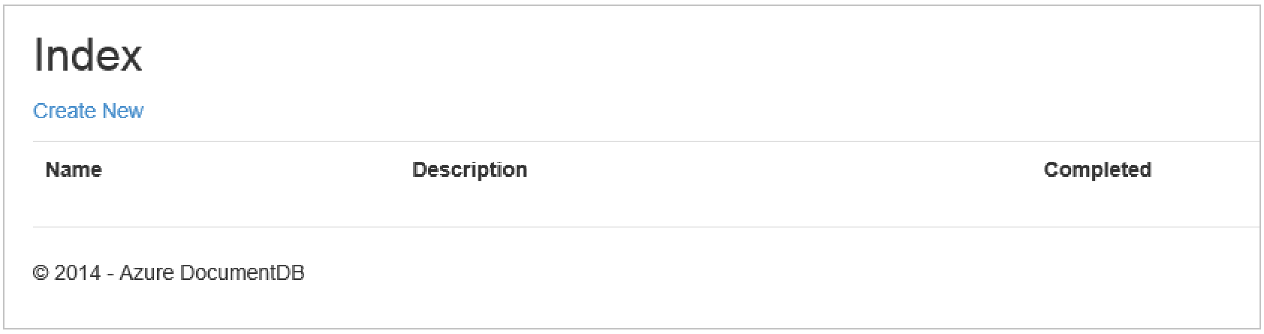
Open ***App_Start\RouteConfig.cs*** and locate the line starting with "defaults:" and change it to resemble the following.

```
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 Azure Cosmos DBRepository and ItemController to persist the record in Azure Cosmos DB.

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 Azure Cosmos DB 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 Azure Cosmos DB 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 Azure Cosmos DB and passes it to the **Edit** view.

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

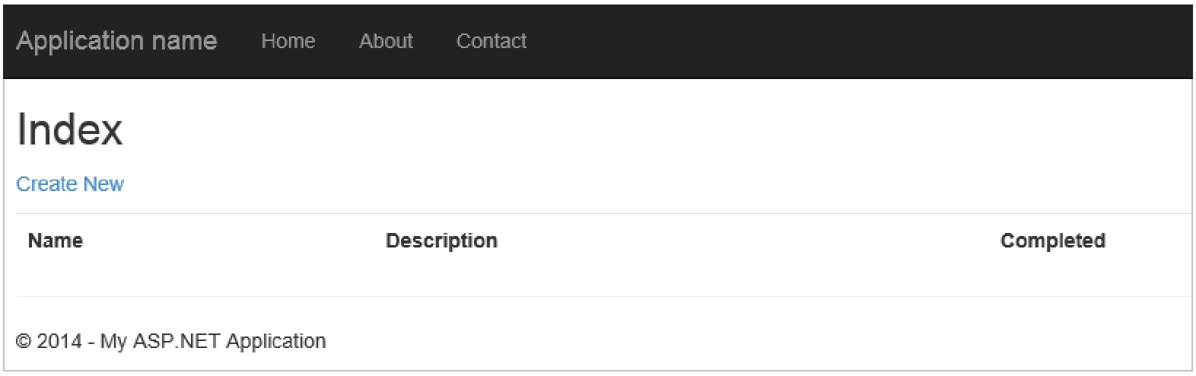
The second method we added handles passing the updated object to Azure Cosmos DB to be persisted in the database.

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:



Index

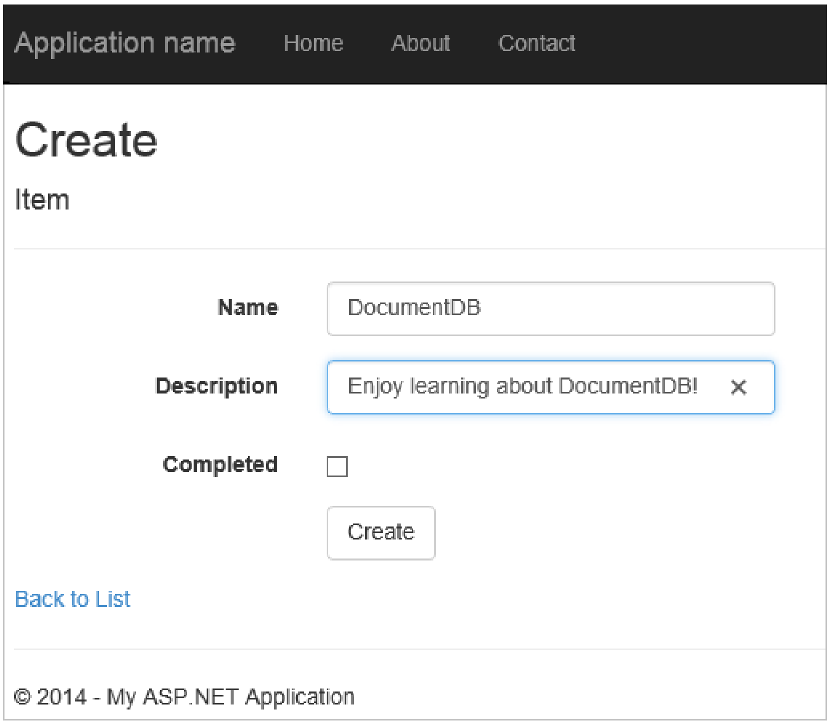
Create New

Name	Description	Completed

© 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.



Create

Item

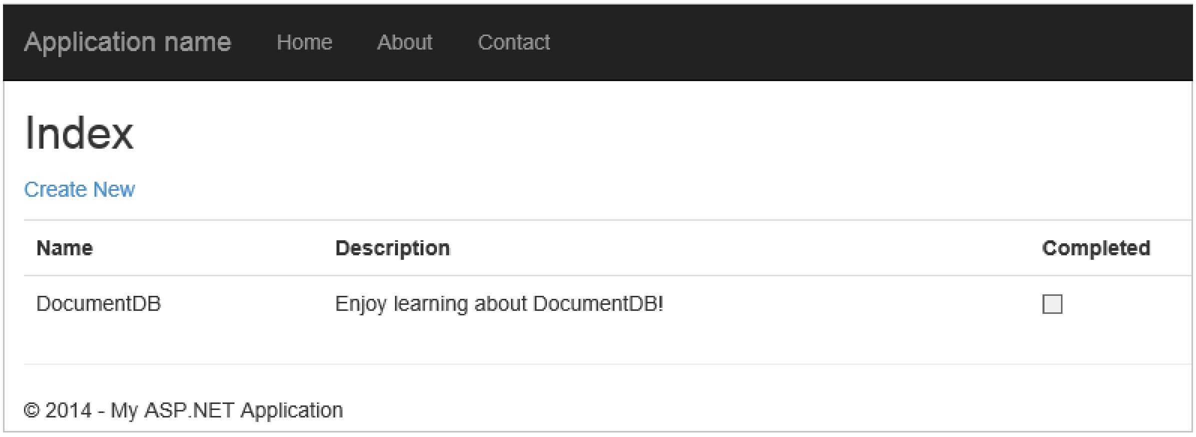
Name: DocumentDB

Description: Enjoy learning about DocumentDB!

Completed: □

© 2014 - My ASP.NET Application

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



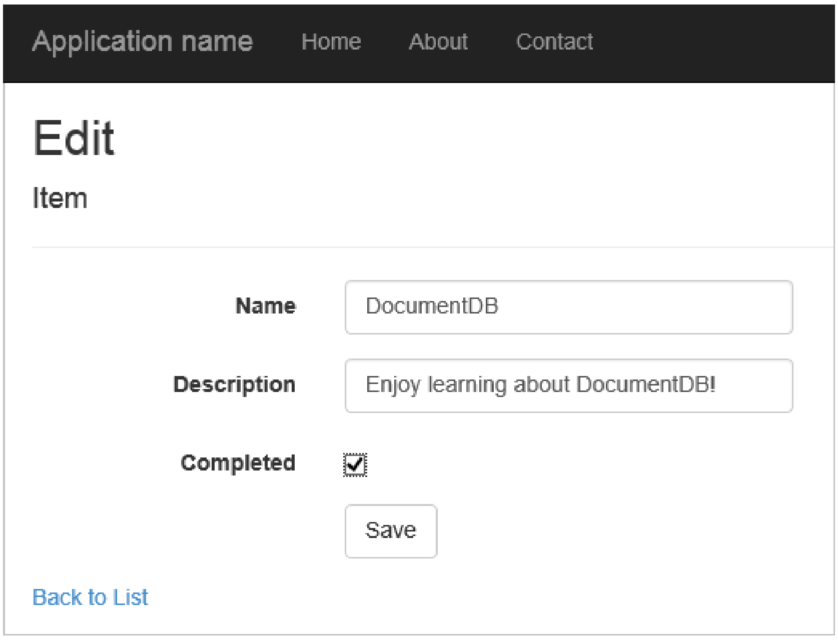
Index

Name	Description	Completed
DocumentDB	Enjoy learning about DocumentDB!	□

© 2014 - My ASP.NET Application

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

4. 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.



Edit

Item

Name: DocumentDB

Description: Enjoy learning about DocumentDB!

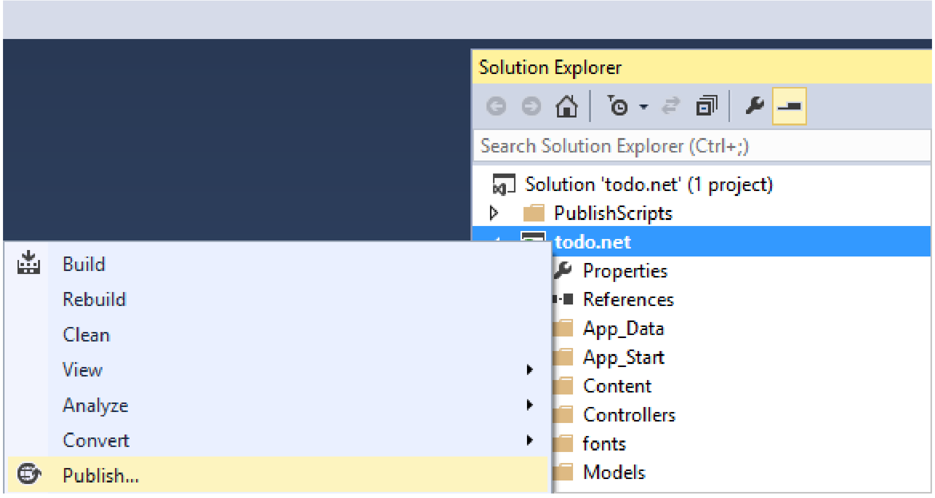
Completed: ☑

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

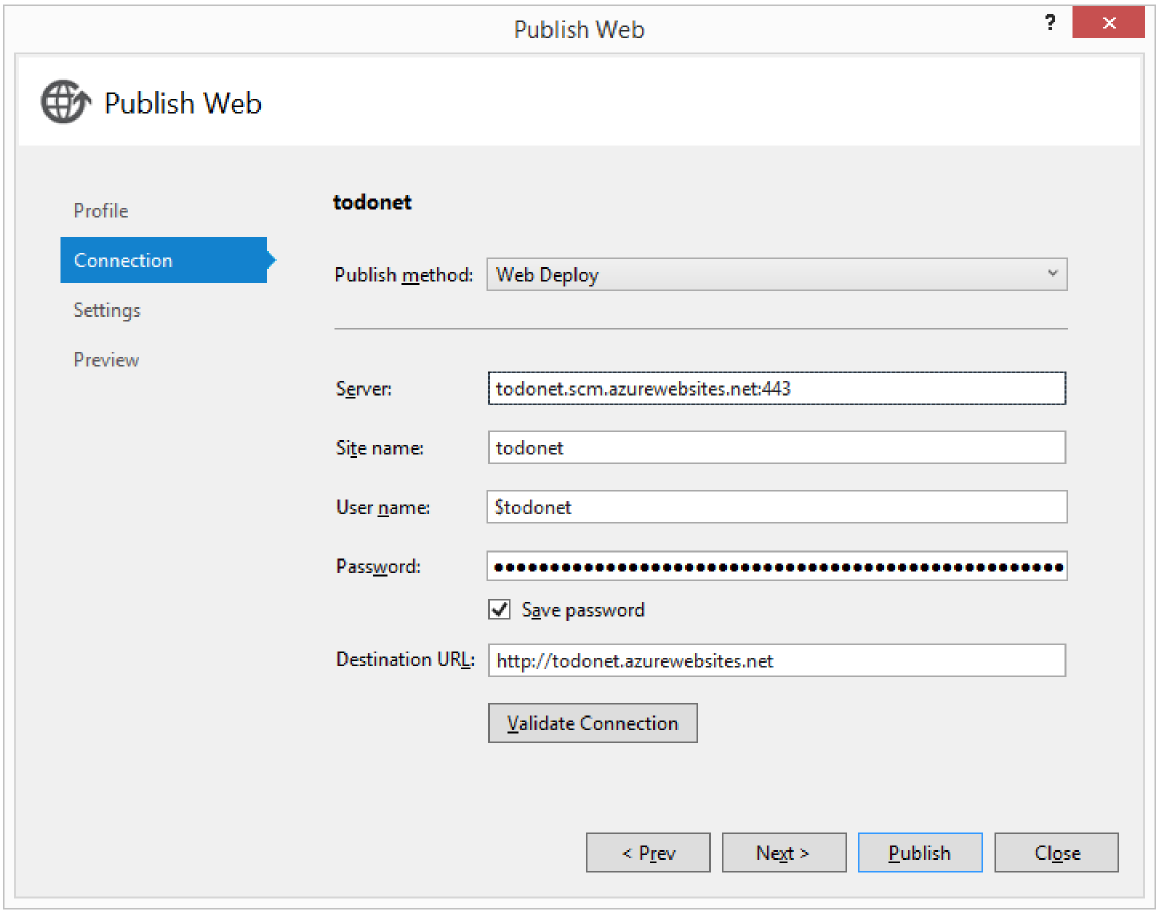
Step 7: Deploy the application to Azure Websites

Now that you have the complete application working correctly with Azure Cosmos DB 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.

1. 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**.



todonet

Publish method: Web Deploy

Server: todonet.scm.azurewebsites.net:443

Site name: todonet

User name: $todonet

Password: ●●

☑ Save password

Destination URL: http://todonet.azurewebsites.net

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 Cosmos DB 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 mobile applications with Xamarin and Azure Cosmos DB

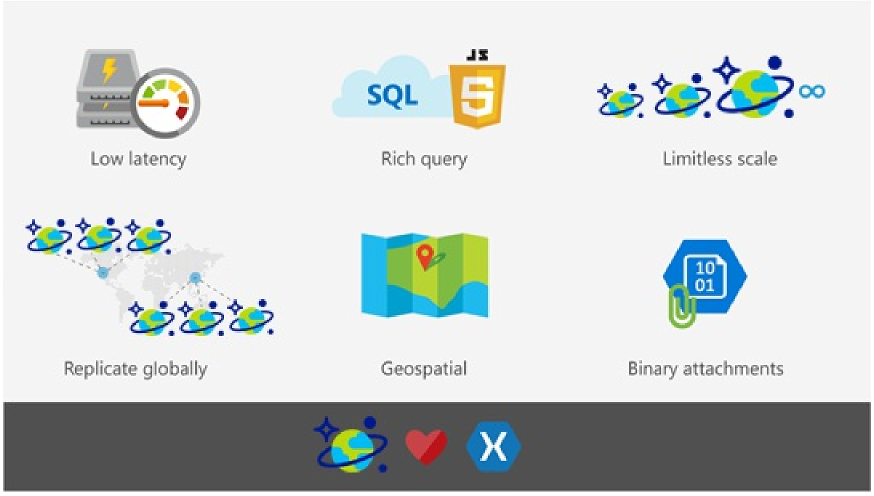
5/30/2017 • 5 min to read • Edit Online

Most mobile apps need to store data in the cloud, and Azure Cosmos DB is a cloud database for mobile apps. It has everything a mobile developer needs. It is a fully managed database as a service that scales on demand. It can bring your data to your application transparently, wherever your users are located around the globe. By using the Azure Cosmos DB .NET Core SDK, you can enable Xamarin mobile apps to interact directly with Azure Cosmos DB, without a middle tier.

This article provides a tutorial for building mobile apps with Xamarin and Azure Cosmos DB. You can find the complete source code for the tutorial at Xamarin and Azure Cosmos DB on GitHub, including how to manage users and permissions.

Azure Cosmos DB capabilities for mobile apps

Azure Cosmos DB provides the following key capabilities for mobile app developers:



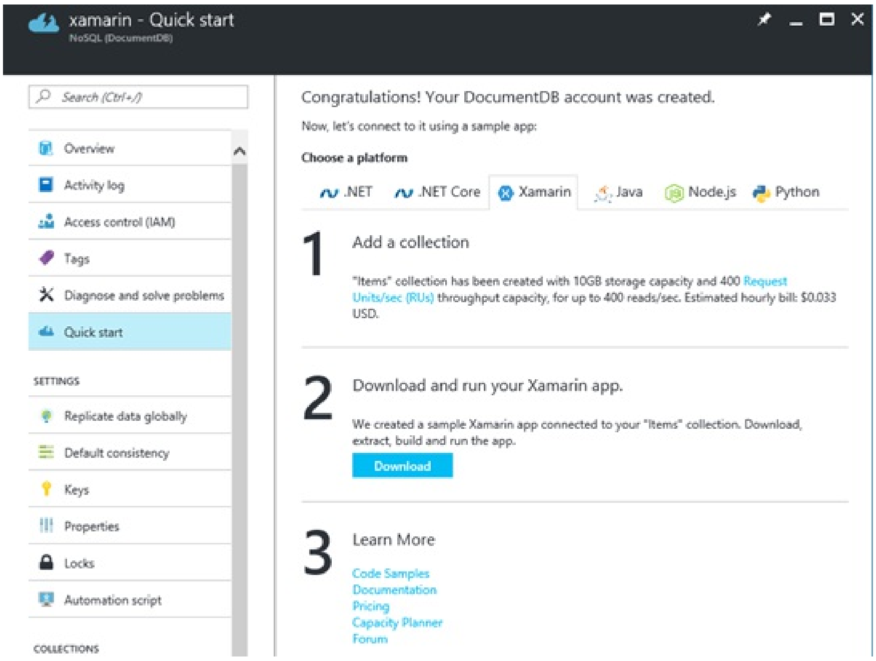
- Rich queries over schemaless data. Azure Cosmos DB stores data as schemaless JSON documents in heterogeneous collections. It offers rich and fast queries without the need to worry about schemas or indexes.
- Fast throughput. It takes only a few milliseconds to read and write documents with Azure Cosmos DB. Developers can specify the throughput they need, and Azure Cosmos DB honors it with 99.99 percent SLAs.
- Limitless scale. Your Azure Cosmos DB collections grow as your app grows. You can start with small data size and throughput of hundreds of requests per second. Your collections can grow to petabytes of data and arbitrarily large throughput with hundreds of millions of requests per second.
- Globally distributed. Mobile app users are on the go, often across the world. Azure Cosmos DB is a globally distributed database. Click the map to make your data accessible to your users.
- Built-in rich authorization. With Azure Cosmos DB, you can easily implement popular patterns like per-user data or multiuser shared data, without complex custom authorization code.
- Geospatial queries. Many mobile apps offer geo-contextual experiences today. With first-class support for geospatial types, DocumentDB makes creating these experiences easy to accomplish.
- Binary attachments. Your app data often includes binary blobs. Native support for attachments makes it easier to use Azure Cosmos DB as a one-stop shop for your app data.

Azure Cosmos DB and Xamarin tutorial

The following tutorial shows how to build a mobile application by using Xamarin and Azure Cosmos DB. You can find the complete source code for the tutorial at Xamarin and Azure Cosmos DB on GitHub.

Get started

It's easy to get started with Azure Cosmos DB. Go to the Azure portal, and create a new Azure Cosmos DB account. Click the **Quick start** tab. Download the Xamarin Forms to-do list sample that is already connected to your Azure Cosmos DB account.



Congratulations! Your DocumentDB account was created.

Now, let's connect to it using a sample app:

Choose a platform

1 Add a collection

"Items" collection has been created with 10GB storage capacity and 400 Request Units/sec (RUs) throughput capacity, for up to 400 reads/sec. Estimated hourly bill: $0.033 USD.

2 Download and run your Xamarin app.

We created a sample Xamarin app connected to your "Items" collection. Download, extract, build and run the app.

3 Learn More

Or if you have an existing Xamarin app, you can add the Azure Cosmos DB NuGet package. Azure Cosmos DB supports Xamarin.IOS, Xamarin.Android, and Xamarin Forms shared libraries.

Work with data

Your data records are stored in Azure Cosmos DB as schemaless JSON documents in heterogeneous collections. You can store documents with different structures in the same collection:

```
var result = await client.CreateDocumentAsync(collectionLink, todoItem);
```

In your Xamarin projects, you can use language-integrated queries over schemaless data:

```
var query = await client.CreateDocumentQuery<ToDoItem>(collectionLink)
            .Where(todoItem => todoItem.Complete == false)
            .AsDocumentQuery();

Items = new List<TodoItem>();
while (query.HasMoreResults) {
   Items.AddRange(await query.ExecuteNextAsync<TodoItem>());
}
```

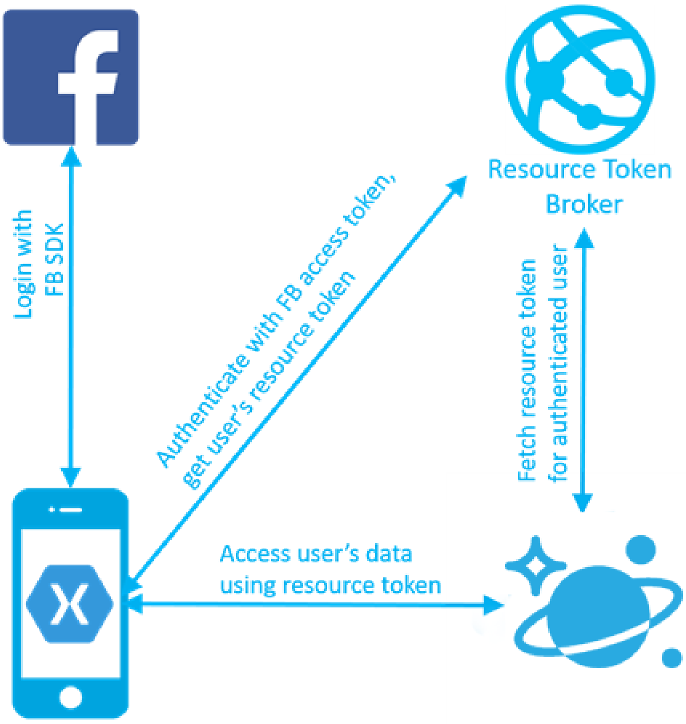
Add users

Like many get started samples, the Azure Cosmos DB sample you downloaded authenticates to the service by using a master key hardcoded in the app's code. This default is not a good practice for an app you intend to run anywhere except on your local emulator. If an unauthorized user obtained the master key, all the data across your Azure Cosmos DB account could be compromised. Instead, you want your app to access only the records for the signed-in user. Azure Cosmos DB allows developers to grant application read or read/write permission to a collection, a set of documents grouped by a partition key, or a specific document.

Follow these steps to modify the to-do list app to a multiuser to-do list app:

1. Add Login to your app by using Facebook, Active Directory, or any other provider.
2. Create an Azure Cosmos DB UserItems collection with **/userId** as the partition key. Specifying the partition key for your collection allows Azure Cosmos DB to scale infinitely as the number of your app users grows, while continuing to offer fast queries.
3. Add Azure Cosmos DB Resource Token Broker. This simple Web API authenticates users and issues short-lived tokens to signed-in users with access only to the documents within their partition. In this example, Resource Token Broker is hosted in App Service.
4. Modify the app to authenticate to Resource Token Broker with Facebook, and request the resource tokens for the signed-in Facebook users. You can then access their data in the UserItems collection.

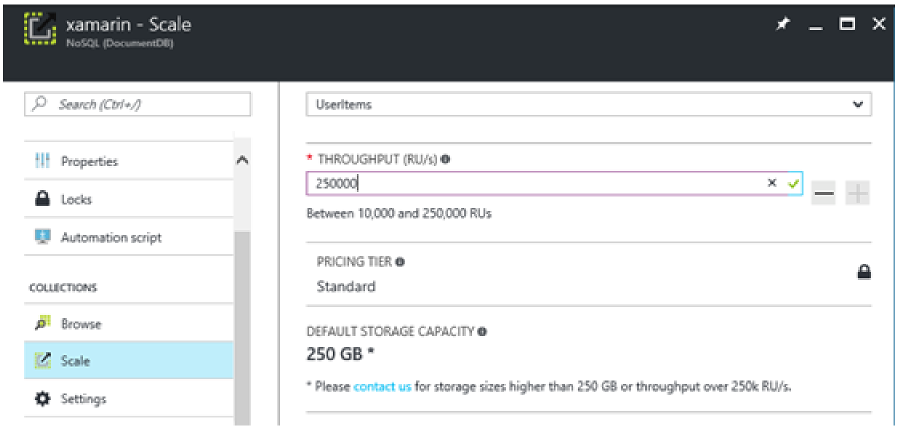
You can find a complete code sample of this pattern at Resource Token Broker on GitHub. This diagram illustrates the solution:


Resource Token Broker
Login with FB SDK
Authenticate with FB access token, get user's resource token
Fetch resource token for authenticated user
Access user's data using resource token

If you want two users to have access to the same to-do list, you can add additional permissions to the access token in Resource Token Broker.

Scale on demand

Azure Cosmos DB is a managed database as a service. As your user base grows, you don't need to worry about provisioning VMs or increasing cores. All you need to tell Azure Cosmos DB is how many operations per second (throughput) your app needs. You can specify the throughput via the **Scale** tab by using a measure of throughput called Request Units (RUs) per second. For example, a read operation on a 1-KB document requires 1 RU. You can also add alerts to the **Throughput** metric to monitor the traffic growth and programmatically change the throughput as alerts fire.


xamarin - Scale
NoSQL (DocumentDB)

UserItems

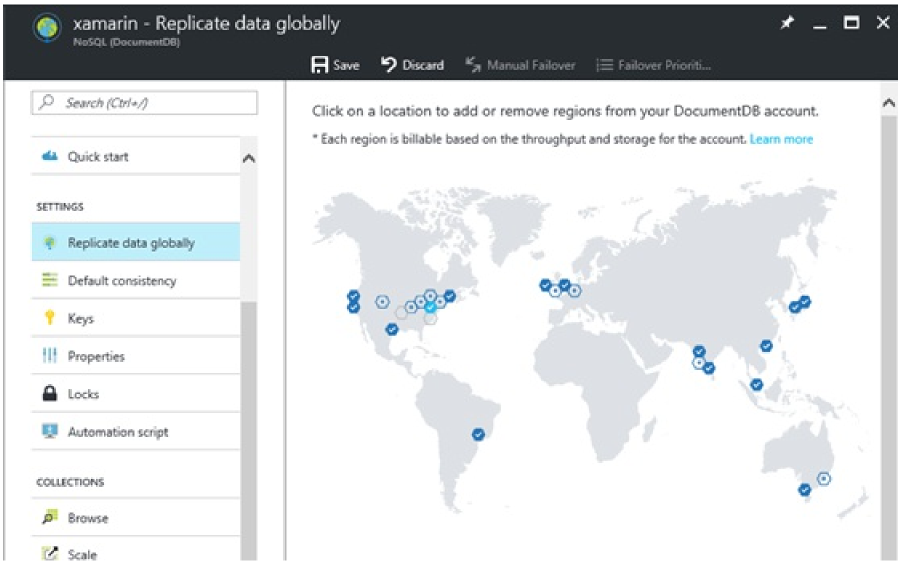
* THROUGHPUT (RU/s)
250000
Between 10,000 and 250,000 RUs

PRICING TIER
Standard

DEFAULT STORAGE CAPACITY
250 GB *
* Please contact us for storage sizes higher than 250 GB or throughput over 250k RU/s.

Go planet scale

As your app gains popularity, you might gain users across the globe. Or maybe you want to be prepared for unforeseen events. Go to the Azure portal, and open your Azure Cosmos DB account. Click the map to make your data continuously replicate to any number of regions across the world. This capability makes your data available wherever your users are. You can also add failover policies to be prepared for contingencies.


xamarin - Replicate data globally
NoSQL (DocumentDB)

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

Congratulations. You have completed the solution and have a mobile app with Xamarin and Azure Cosmos DB. Follow similar steps to build Cordova apps by using the Azure Cosmos DB JavaScript SDK and native iOS/Android apps by using Azure Cosmos DB REST APIs.

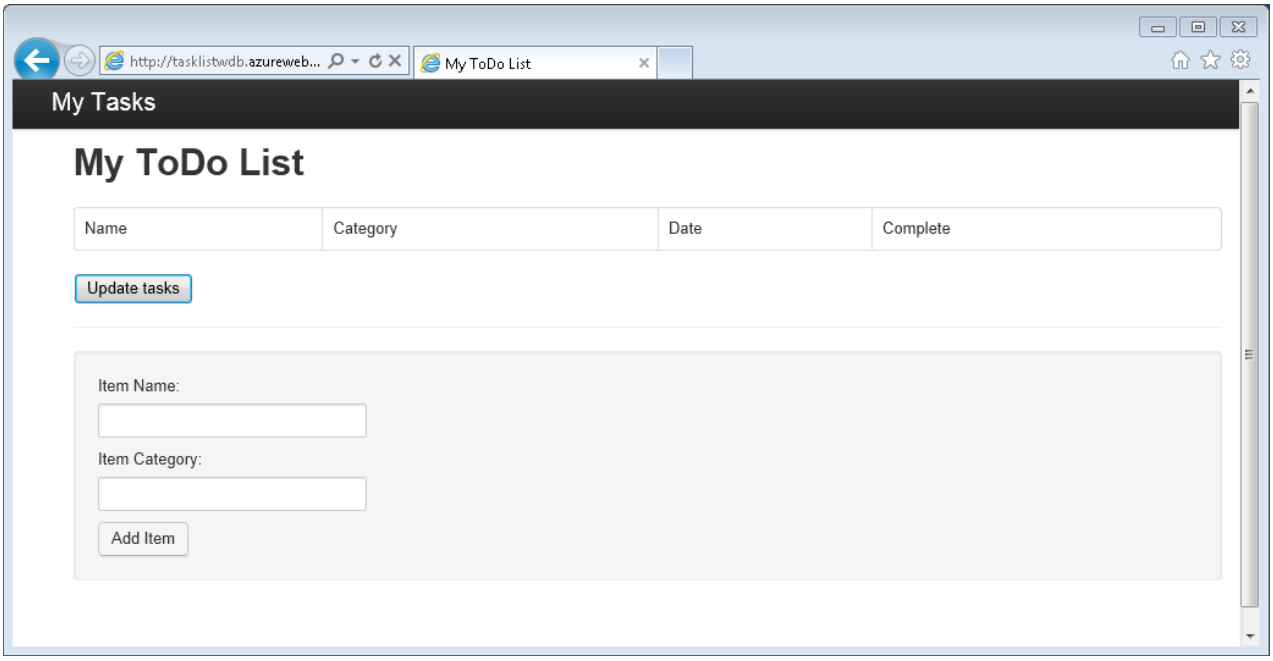
Next steps

- View the source code for Xamarin and Azure Cosmos DB on GitHub.
- Download the DocumentDB .NET Core SDK.
- Find more code samples for .NET applications.
- Learn about Azure Cosmos DB rich query capabilities.
- Learn about geospatial support in Azure Cosmos DB.

Build a Node.js web application using Azure Cosmos DB

5/30/2017 • 14 min to read • Edit Online

This Node.js tutorial shows you how to use Azure Cosmos DB and the DocumentDB API to store and access data from a Node.js Express application hosted on Azure Websites. You build a simple web-based task-management application, a ToDo app, that allows creating, retrieving, and completing tasks. The tasks are stored as JSON documents in Azure Cosmos DB. This tutorial walks you through the creation and deployment of the app and explains what's happening in each snippet.



My ToDo List

Name	Category	Date	Complete

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. Just read the Readme file for instructions on how to run the app.

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.

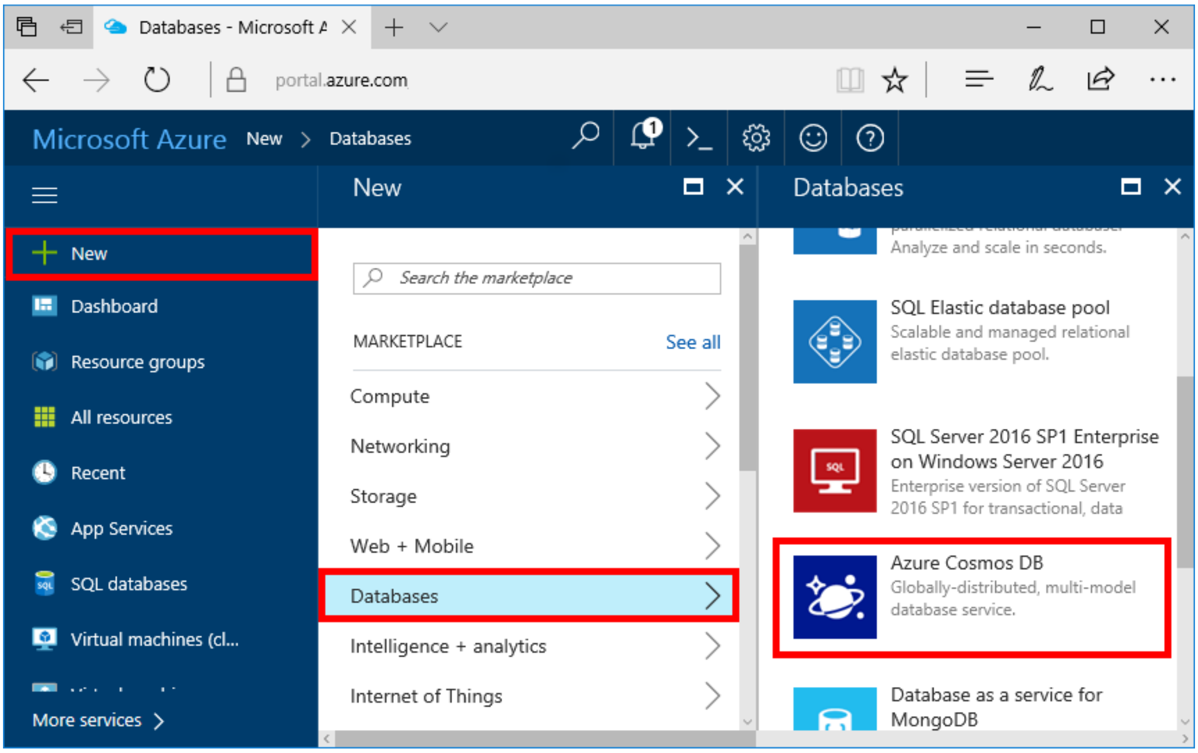
  OR

  A local installation of the Azure Cosmos DB 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 an Azure Cosmos DB database account

Let's start by creating an Azure Cosmos DB account. If you already have an account or if you are using the Azure Cosmos DB Emulator for this tutorial, you can skip to Step 2: Create a new Node.js application.

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

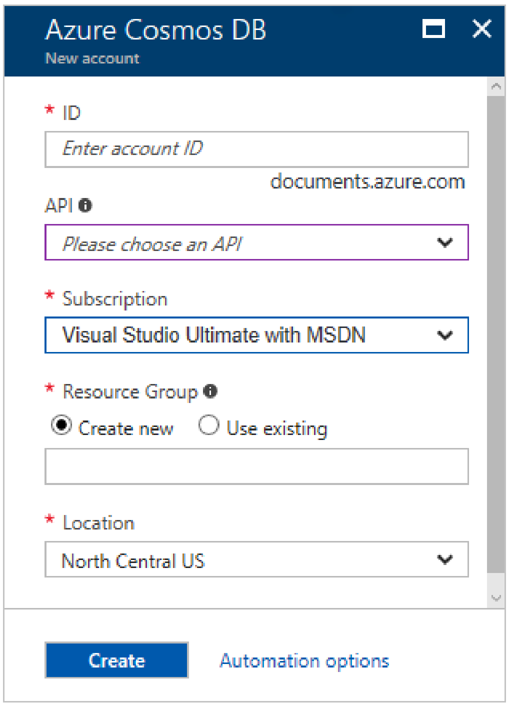


3. On the **New account** blade, specify the configuration that you want for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article we program against the DocumentDB API, so choose **SQL (DocumentDB)** as you fill out the form. But if you have graph data for a social media app, or key/value (table) data, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

   Complete the fields on the **New account** blade, using the information in the following screenshot as a guide. When you set up your account, choose unique values that do not match those in the screenshot.



Azure Cosmos DB

New account

* ID

Enter account ID

documents.azure.com

API

Please choose an API

* Subscription

Visual Studio Ultimate with MSDN

* Resource Group

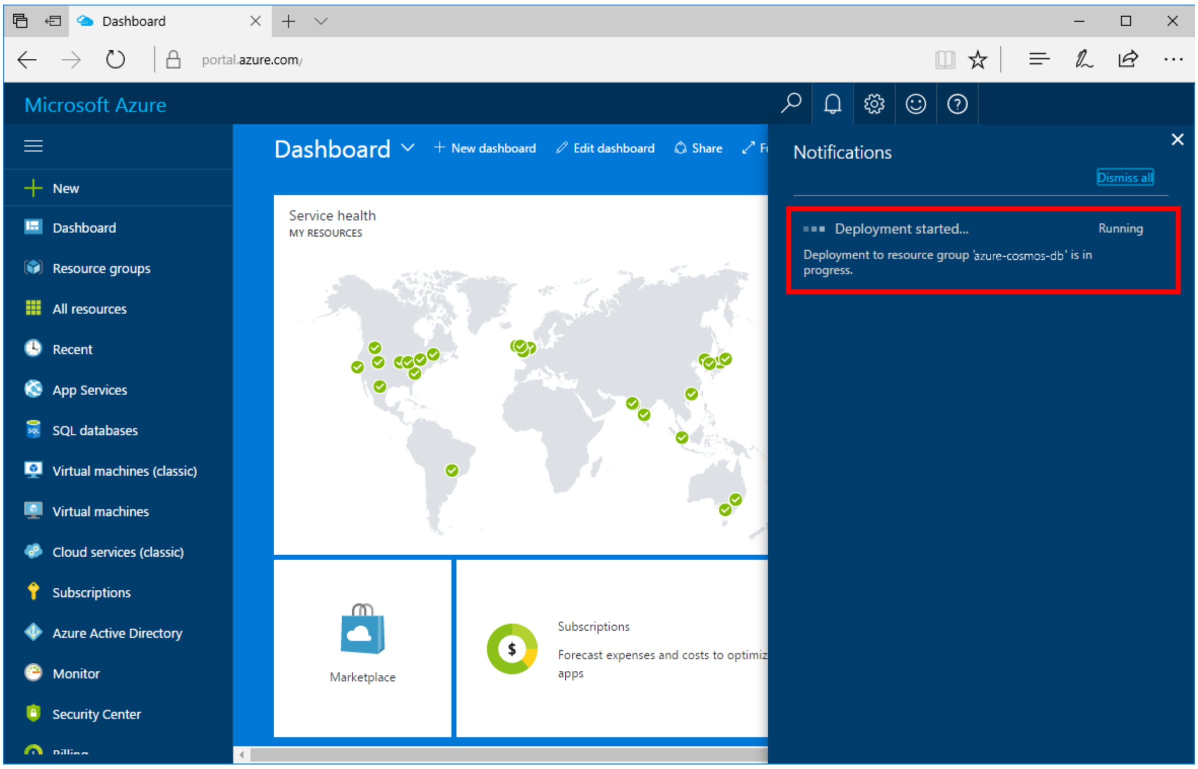
● Create new ○ Use existing

* Location

North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that identifies your Azure Cosmos DB account. The string *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 through 50 characters.
API	SQL (DocumentDB)	We program against the DocumentDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for your Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource-group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location that's closest to your users to give them the fastest access to the data.

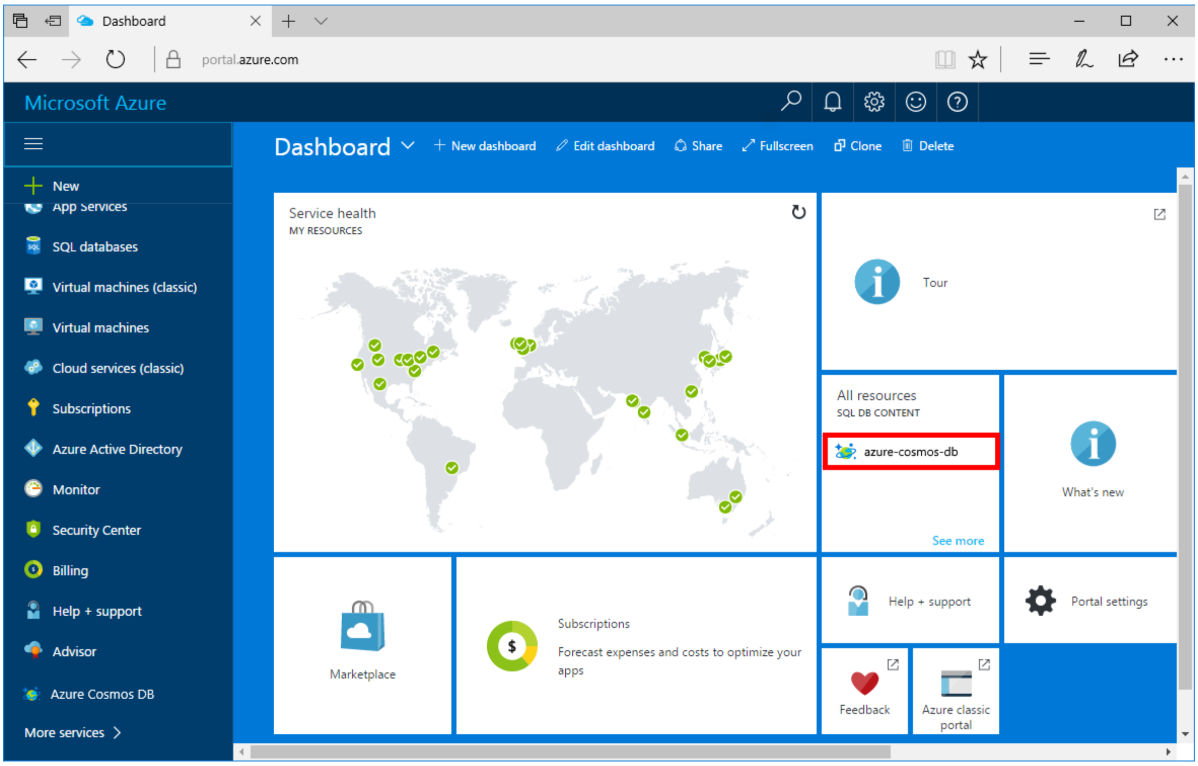
4. Click **Create** to create the account.
5. On the top toolbar, click **Notifications** to monitor the deployment process.



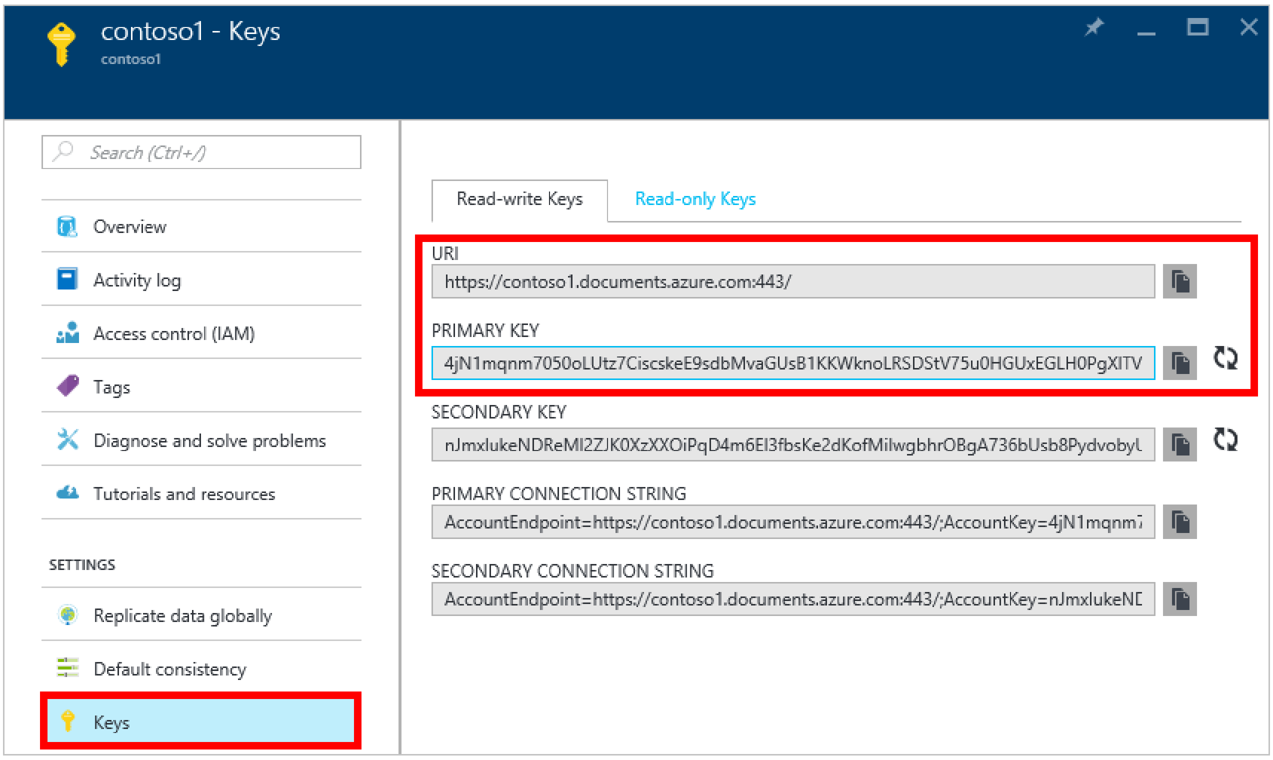
Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the **All Resources** tile.



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



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=nJmxlukeND

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, such as the Node.js command prompt.
2. Navigate to the directory in which you'd like to store the new application.
3. Use the express generator to generate a new application called **todo**.

```
express todo
```

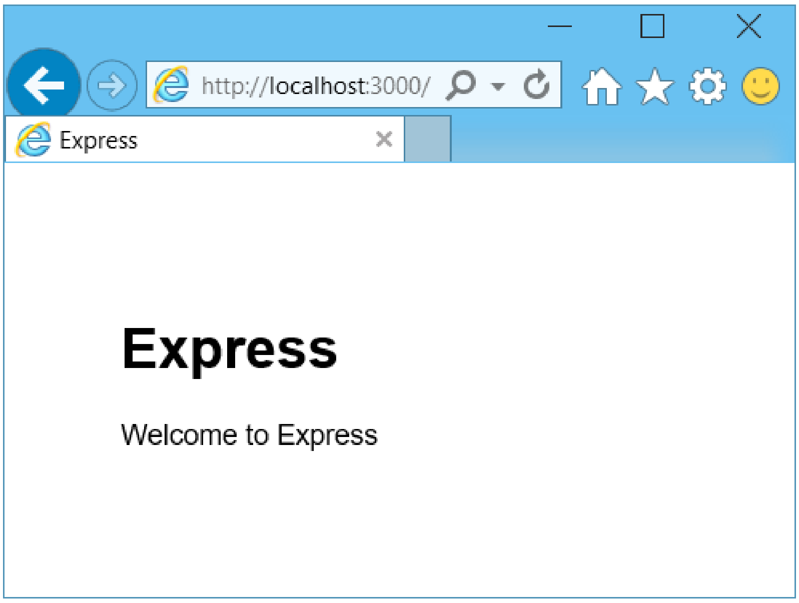
4. Open your new **todo** directory and install dependencies.

```
cd todo
npm install
```

5. Run your new application.

```
npm start
```

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



Express

Welcome to Express

Then, to stop the application, press CTRL+C in the terminal window and then click **y** to terminate the batch job.

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.

```
{
  "name": "todo",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "async": "^2.1.4",
    "body-parser": "~1.15.2",
    "cookie-parser": "~1.4.3",
    "debug": "~2.2.0",
    "documentdb": "^1.10.0",
    "express": "~4.14.0",
    "jade": "~1.11.0",
    "morgan": "~1.7.0",
    "serve-favicon": "~2.3.0"
  }
}
```

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

Step 4: Using the Azure Cosmos DB 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 Cosmos DB.

Create the model

1. In the project directory, create a new directory named **models** in the same directory as the package.json file.
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 Azure Cosmos DB Offer Types please refer to Performance levels in Azure Cosmos DB

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 Azure Cosmos DB.

```
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 {
            } 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 Azure Cosmos DB 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 into some variables we will use soon.
4. Replace the following two lines in **app.js** file:

```
app.use('/', index);
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 Azure Cosmos DB (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="textbox")
    label Item Category:
    input(name="category", type="textbox")
    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.

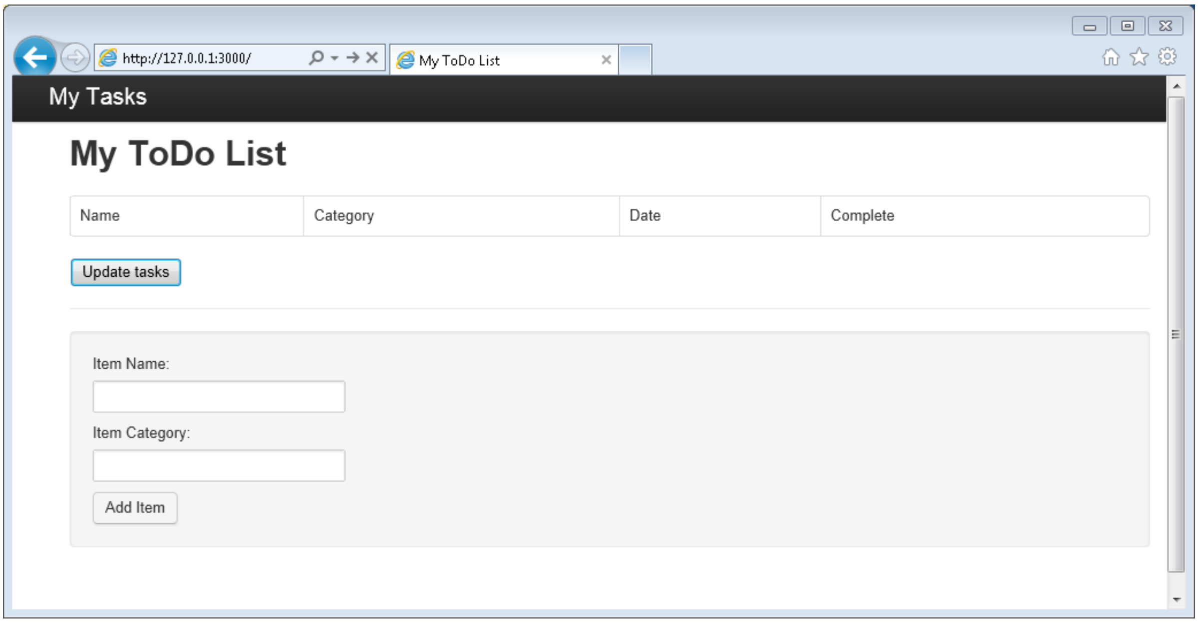
4. Open the **style.css** file in **public\stylesheets** directory and replace the code with the following:

```
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 the terminal to start your application, then refresh your http://localhost:3000 browser page. The page should now look like the image below:



My ToDo List

Name	Category	Date	Complete

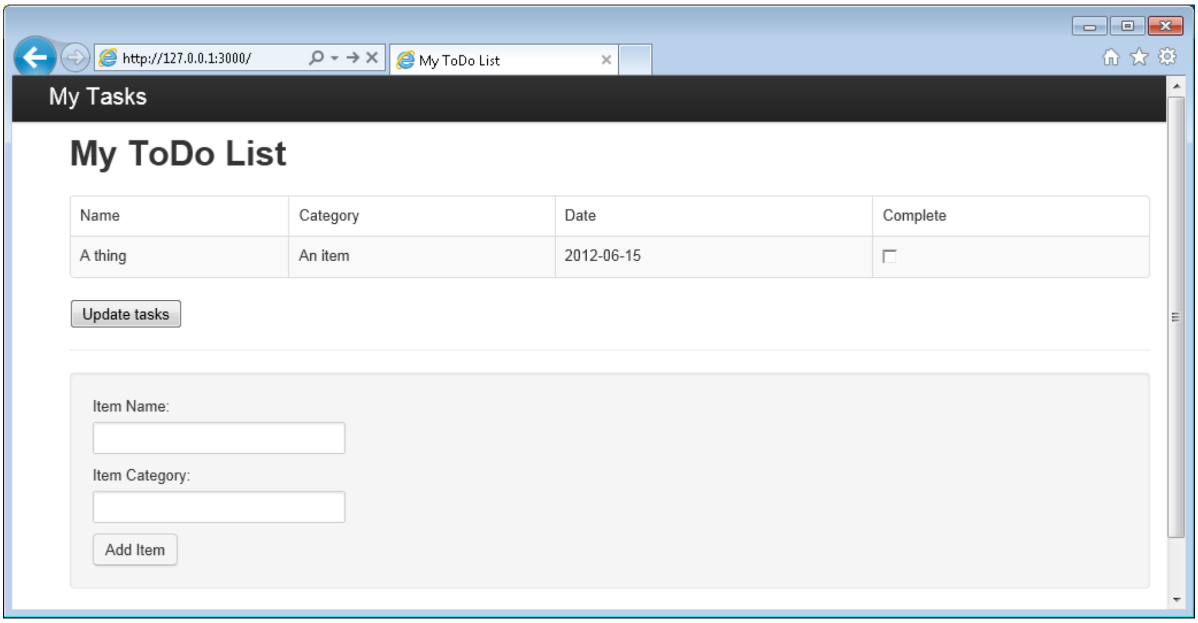
Item Name:

Item Category:

> TIP
>
> If you receive an error about the indent in the layout.jade file or the index.jade file, ensure that the first two lines in both files is left justified, with no spaces. If there are spaces before the first two lines, remove them, save both files, then refresh your browser window.

2. Use the Item, Item Name and Category fields to enter a new task and then click **Add Item**. This creates a document in Azure Cosmos DB with those properties.

3. The page should update to display the newly created item in the ToDo list.



My Tasks

My ToDo List

Name	Category	Date	Complete
A thing	An item	2012-06-15	□

Update tasks

Item Name:

Item Category:

Add Item

4. To complete a task, simply check the checkbox in the Complete column, and then click **Update tasks**. This updates the document you already created.

5. To stop the application, press CTRL+C in the terminal window and then click **Y** to terminate the batch job.

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!

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

   If you want to download or refer to the complete reference application for this tutorial, it can be downloaded from GitHub.

Next steps

- Want to perform scale and performance testing with Azure Cosmos DB? See Performance and Scale Testing with Azure Cosmos DB
- Learn how to monitor an Azure Cosmos DB account.

- Run queries against our sample dataset in the Query Playground.
- Explore the Azure Cosmos DB documentation.

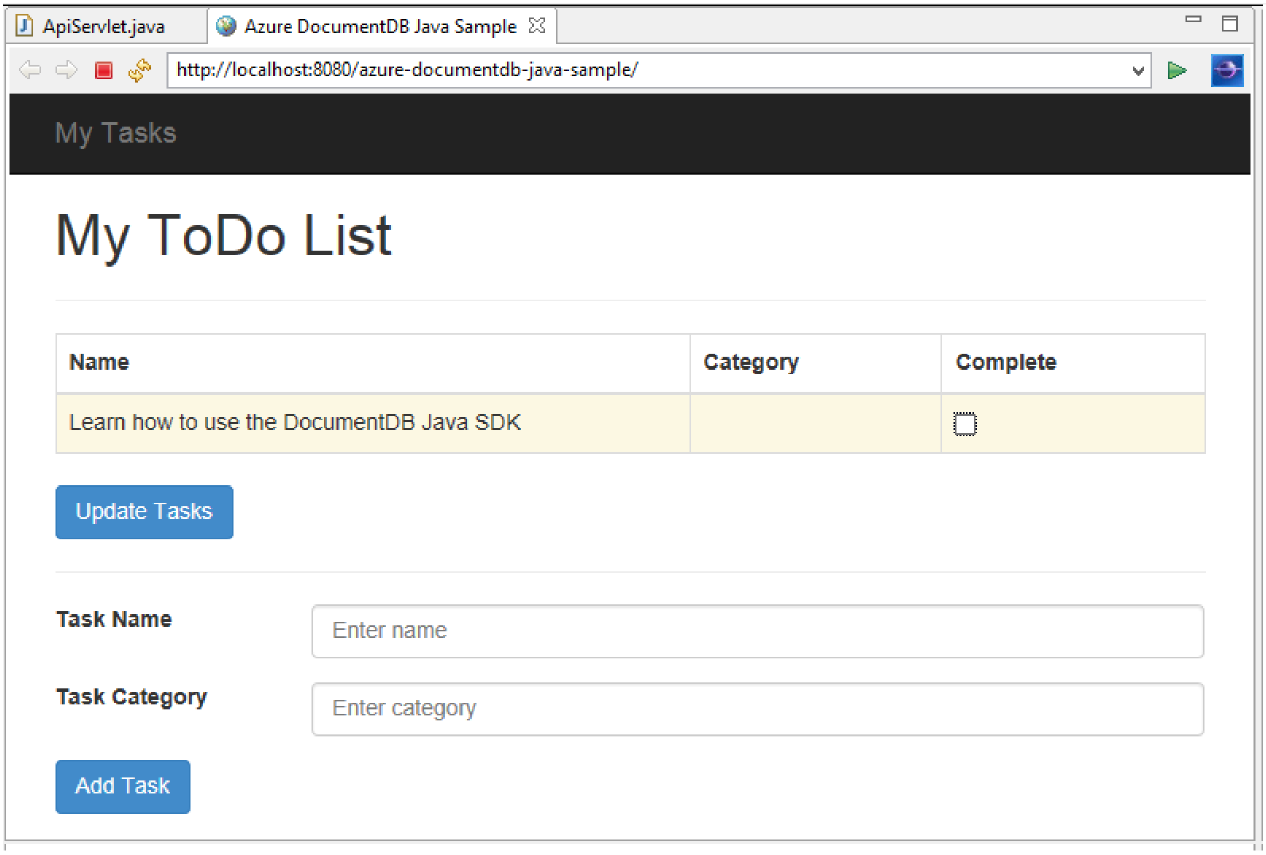
Build a Java web application using Azure Cosmos DB

5/30/2017 • 19 min to read • Edit Online

This Java web application tutorial shows you how to use the Microsoft Azure Cosmos DB 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 Cosmos DB service using the Azure Cosmos DB 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 Cosmos DB.



http://localhost:8080/azure-documentdb-java-sample/

My ToDo List

Name	Category	Complete
Learn how to use the DocumentDB Java SDK		□

Task Name: Enter name

Task Category: Enter category

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 Cosmos DB Emulator.

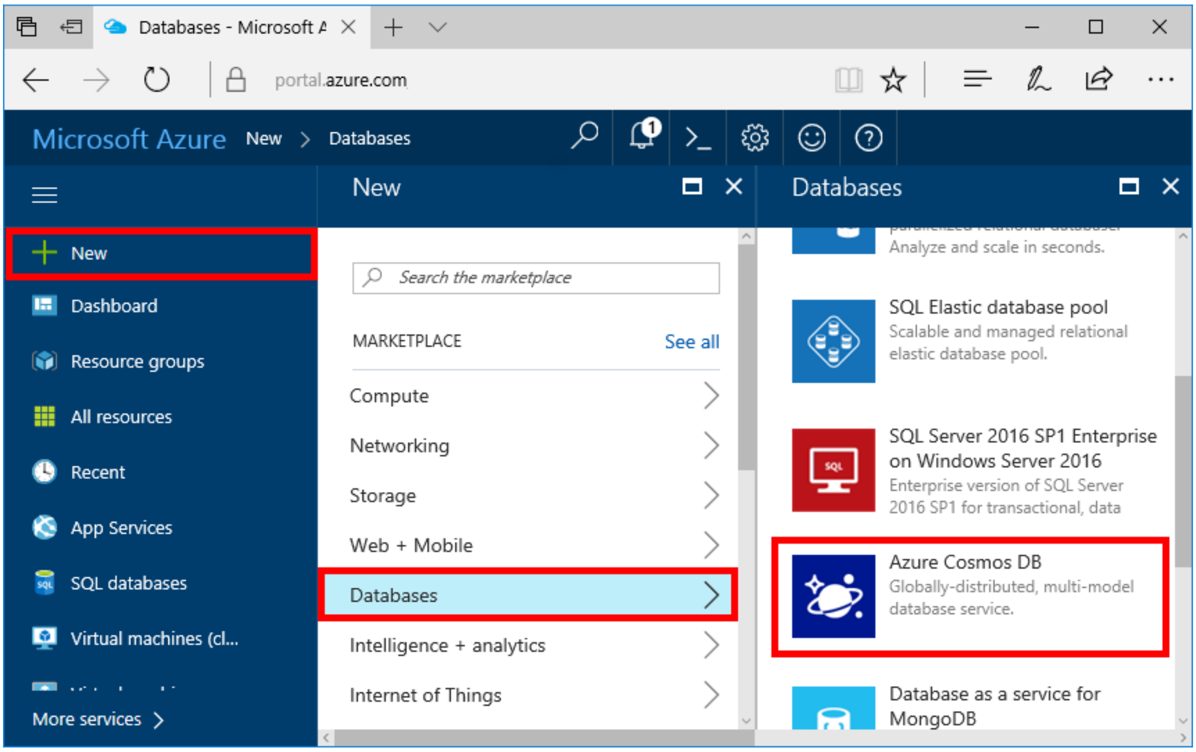
- Java Development Kit (JDK) 7+.
- Eclipse IDE for Java EE Developers.
- An Azure Website with a Java runtime environment (e.g. Tomcat or Jetty) enabled.

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 an Azure Cosmos DB database account

Let's start by creating an Azure Cosmos DB account. If you already have an account or if you are using the Azure Cosmos DB Emulator for this tutorial, you can skip to Step 2: Create the Java JSP application.

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

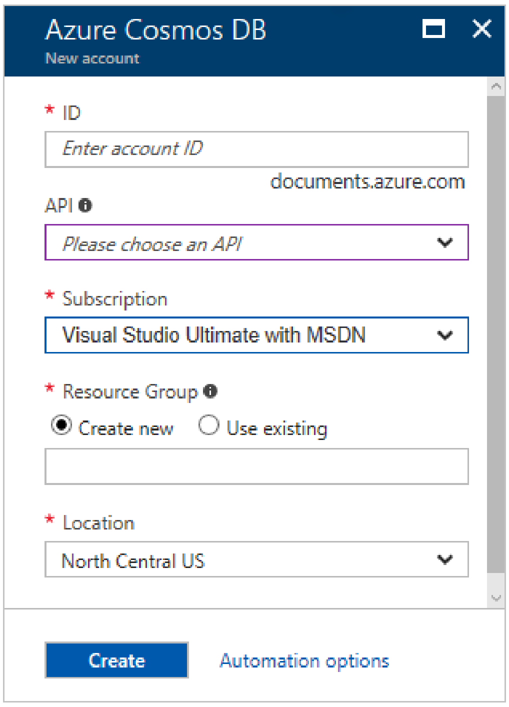


3. On the **New account** blade, specify the configuration that you want for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article we program against the DocumentDB API, so choose **SQL (DocumentDB)** as you fill out the form. But if you have graph data for a social media app, or key/value (table) data, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

   Complete the fields on the **New account** blade, using the information in the following screenshot as a guide. When you set up your account, choose unique values that do not match those in the screenshot.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

* Subscription
Visual Studio Ultimate with MSDN

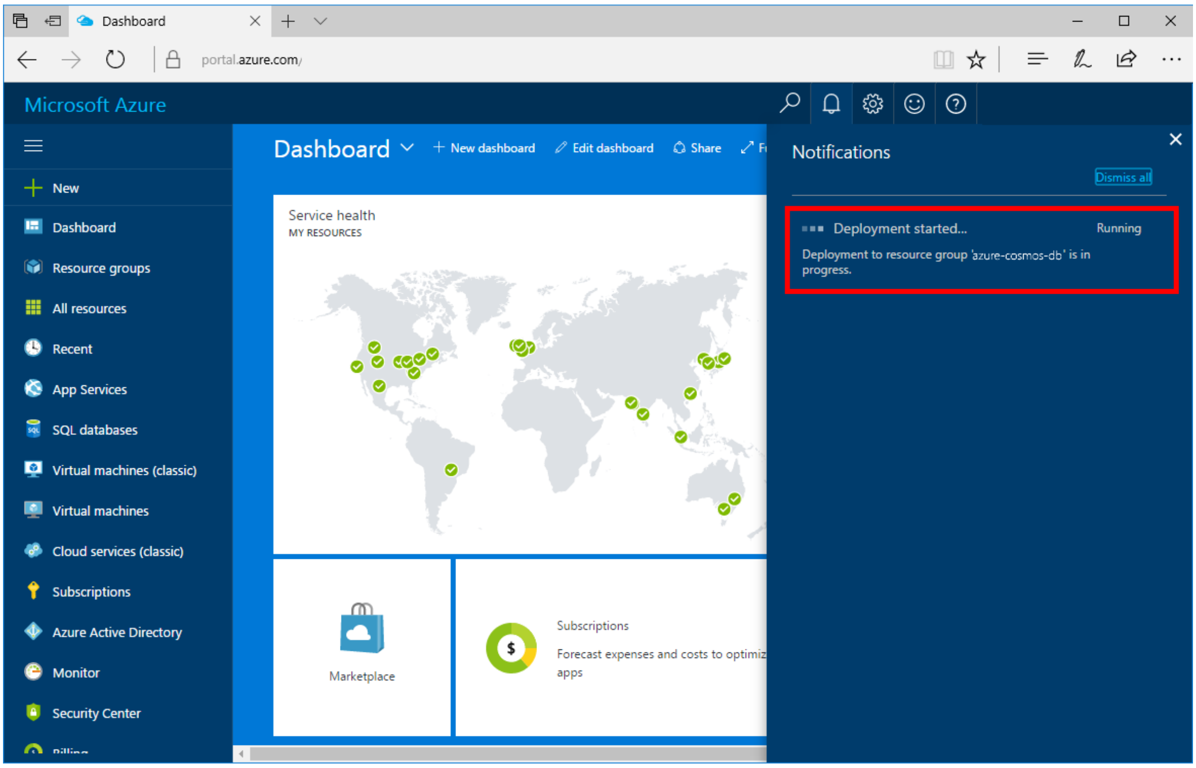
* Resource Group
● Create new ○ Use existing

* Location
North Central US

Create Automation options

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that identifies your Azure Cosmos DB account. The string *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 through 50 characters.
API	SQL (DocumentDB)	We program against the DocumentDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for your Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource-group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location that's closest to your users to give them the fastest access to the data.

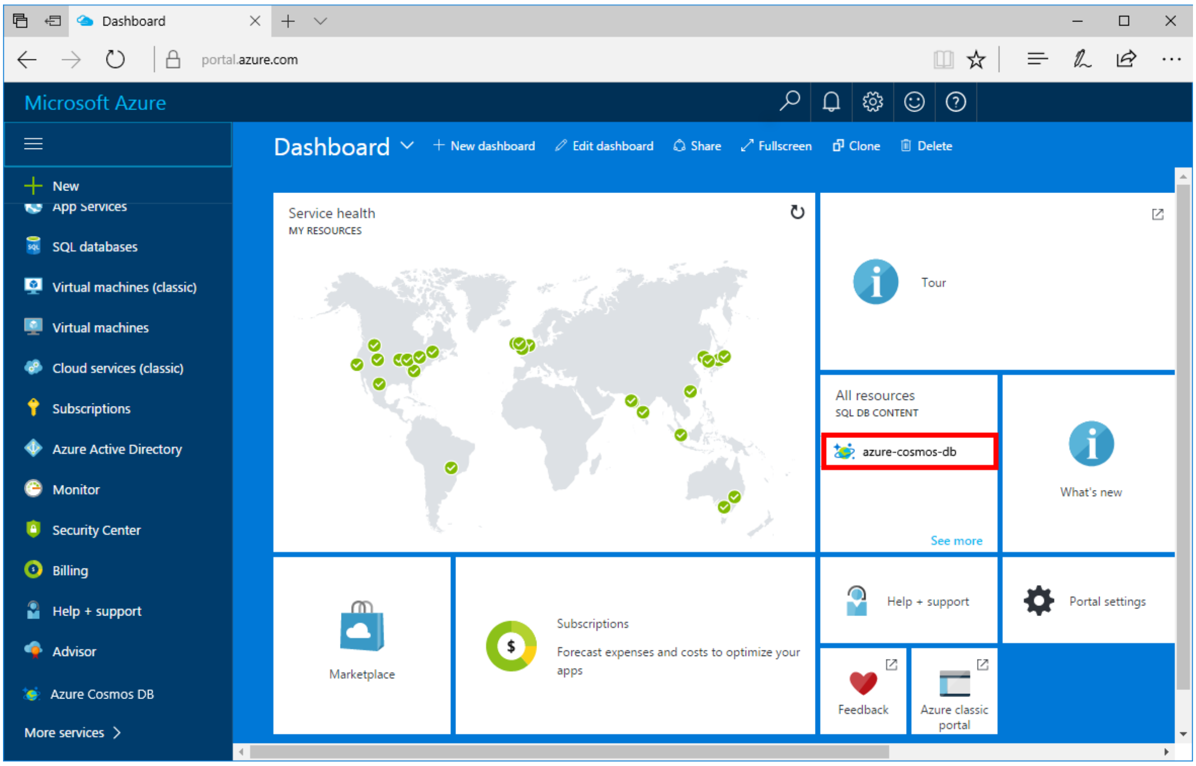
4. Click **Create** to create the account.
5. On the top toolbar, click **Notifications** to monitor the deployment process.



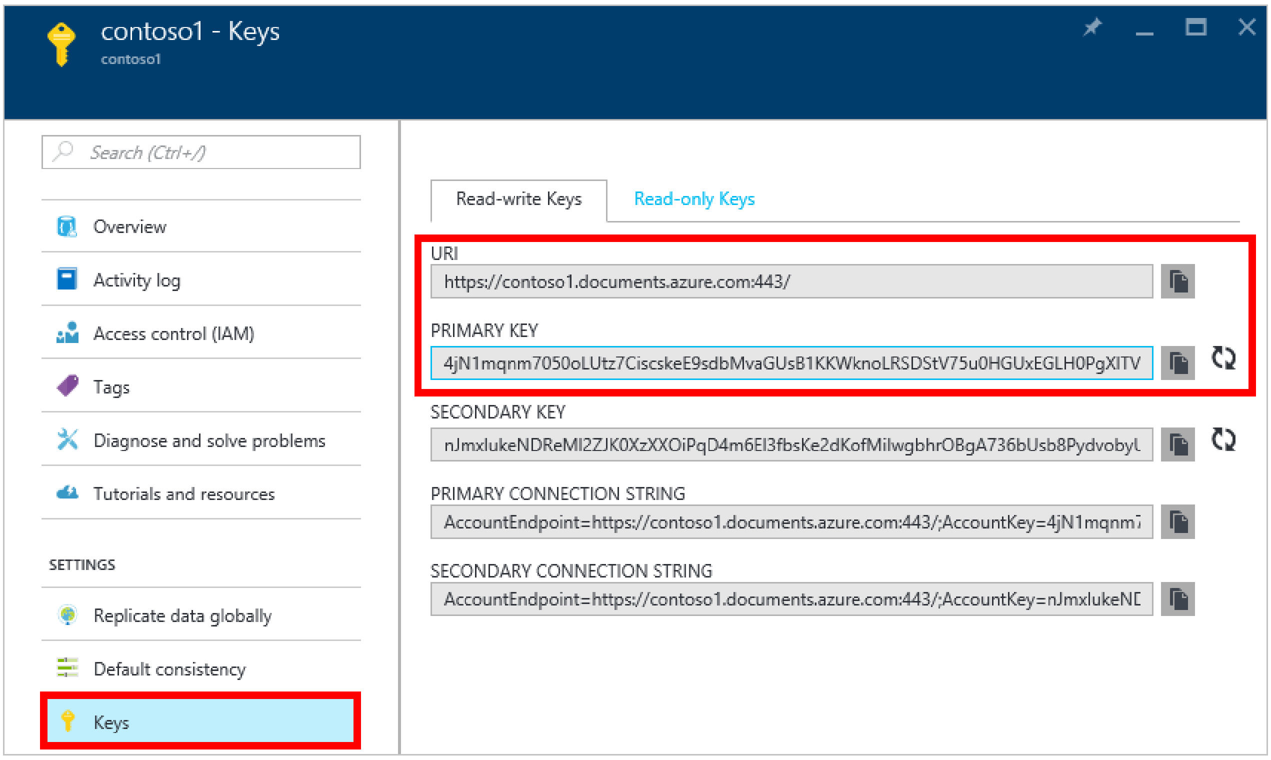
Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the **All Resources** tile.



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



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

PRIMARY KEY
4jN1mqnm7050oLUtz7CiscskeE9sdbMvaGUsB1KKWknoLRSDStV75u0HGUxEGLH0PgXITV

SECONDARY KEY
nJmxlukeNDReMI2ZJK0XzXXOiPqD4m6EI3fbsKe2dKofMilwgbhrOBgA736bUsb8Pydvoby

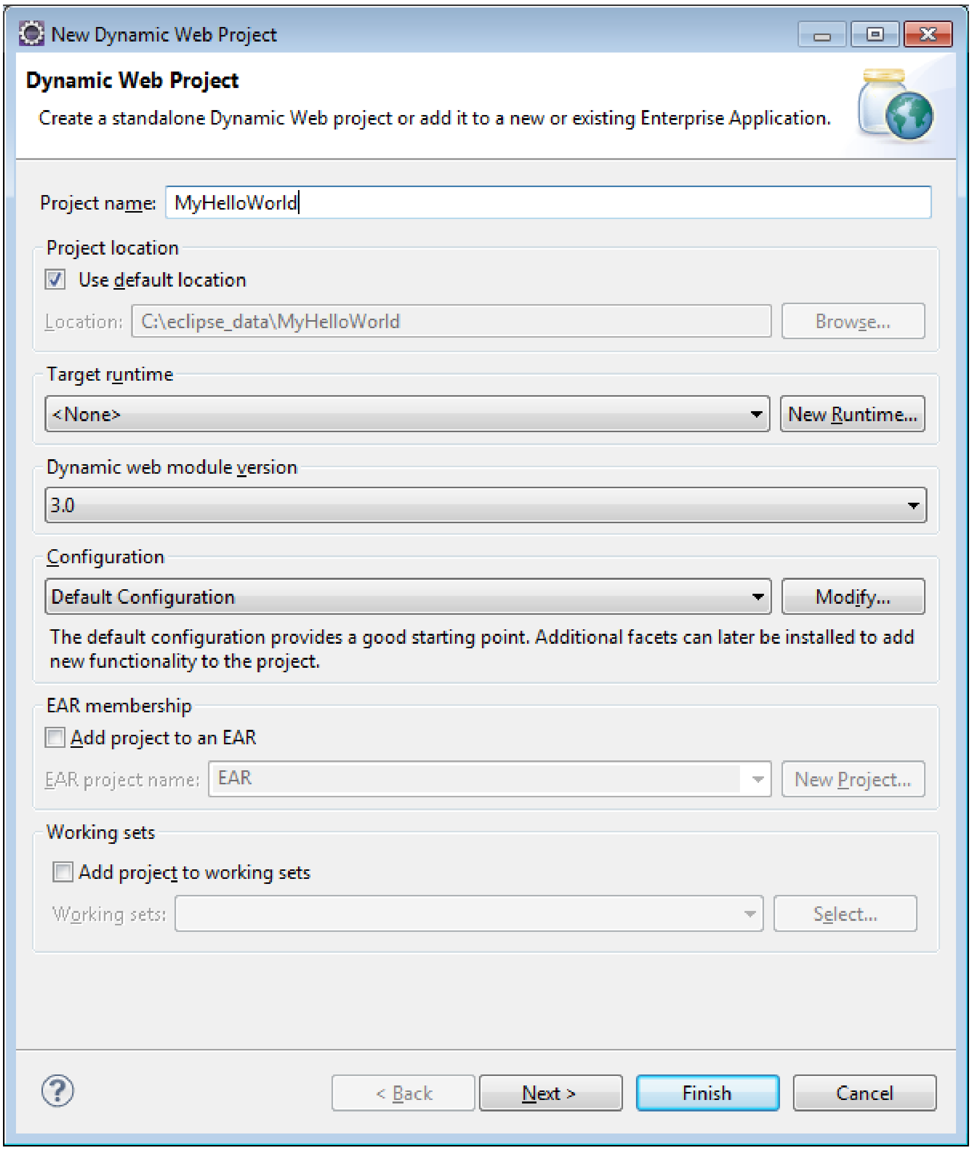
PRIMARY CONNECTION STRING
AccountEndpoint=https://contoso1.documents.azure.com:443/;AccountKey=4jN1mqnm7

SECONDARY CONNECTION STRING
AccountEndpoint=https://contoso1.documents.azure.com:443/;AccountKey=nJmxlukeND

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

Target runtime: <None>

Dynamic web module version: 3.0

Configuration: Default Configuration

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

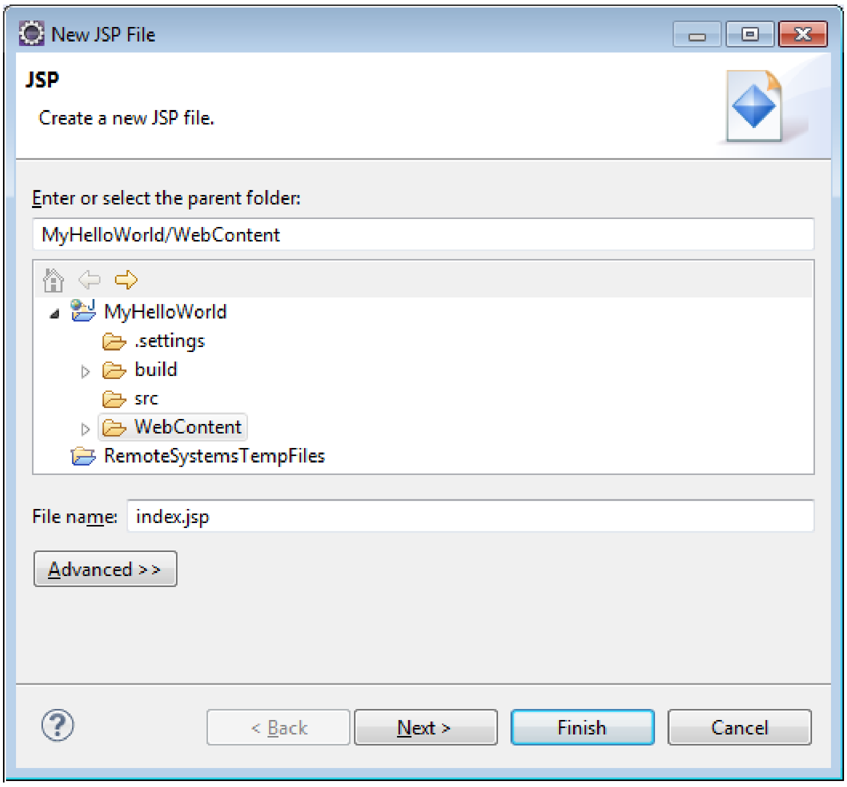
EAR project name: EAR

Working sets

☐ Add project to working sets

Working sets:

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**.



JSP

Create a new JSP file.

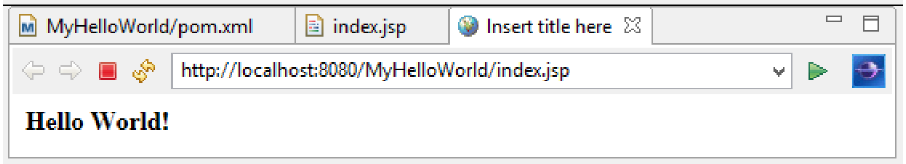
Enter or select the parent folder: MyHelloWorld/WebContent

File name: index.jsp

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:



http://localhost:8080/MyHelloWorld/index.jsp

Hello World!

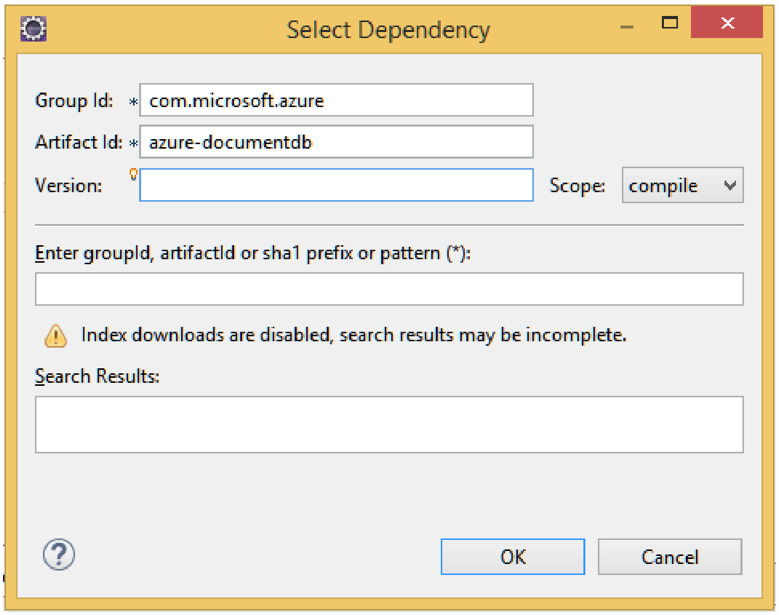
Step 3: Install the DocumentDB Java SDK

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.



Group Id: * com.microsoft.azure

Artifact Id: * azure-documentdb

Version:

Scope: compile

Enter groupId, artifactId or sha1 prefix or pattern (*):

Index downloads are disabled, search results may be incomplete.

Search Results:

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 Azure Cosmos DB 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 Azure Cosmos DB 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 Azure Cosmos DB.

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 Azure Cosmos DB 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
        // 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.

```
// 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 Azure Cosmos DB 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) {
    // Azure Cosmos DB 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 Cosmos DB 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>

         <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-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/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 **Azure Cosmos DB** project, and then click **Finish**. The Azure Cosmos DB project contains the Azure Cosmos DB 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 Azure Cosmos DB account, and then save the file. For more information, see Step 1. Create an Azure Cosmos DB 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.

Build a Python Flask web application using Azure Cosmos DB

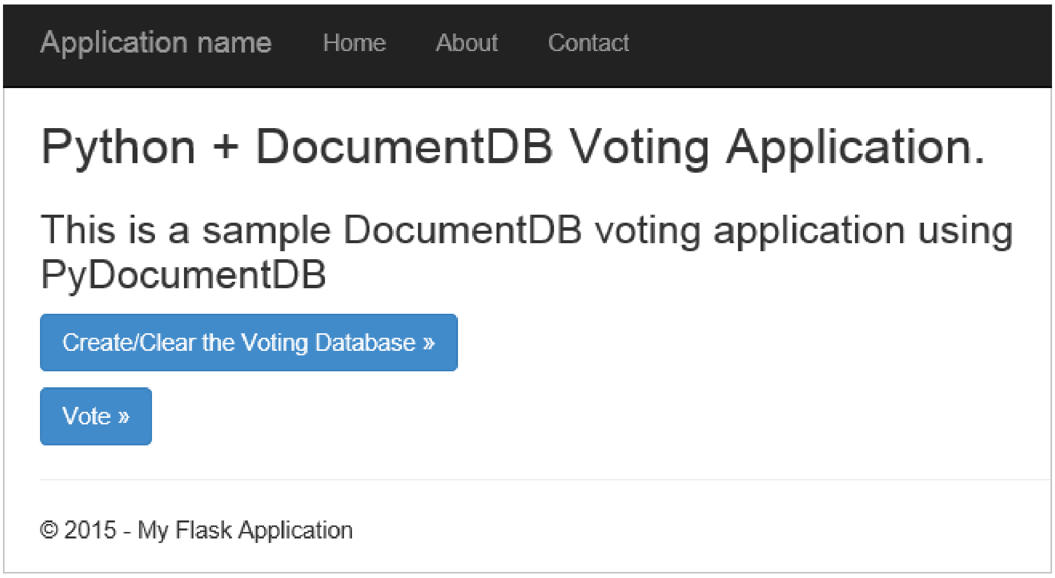
5/30/2017 • 12 min to read • Edit Online

This tutorial shows you how to use Azure Cosmos DB 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 an Cosmos DB account.
2. Creating a Python MVC application.
3. Connecting to and using Cosmos DB 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.



Python + DocumentDB Voting Application.

This is a sample DocumentDB voting application using PyDocumentDB

© 2015 - My Flask Application

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.

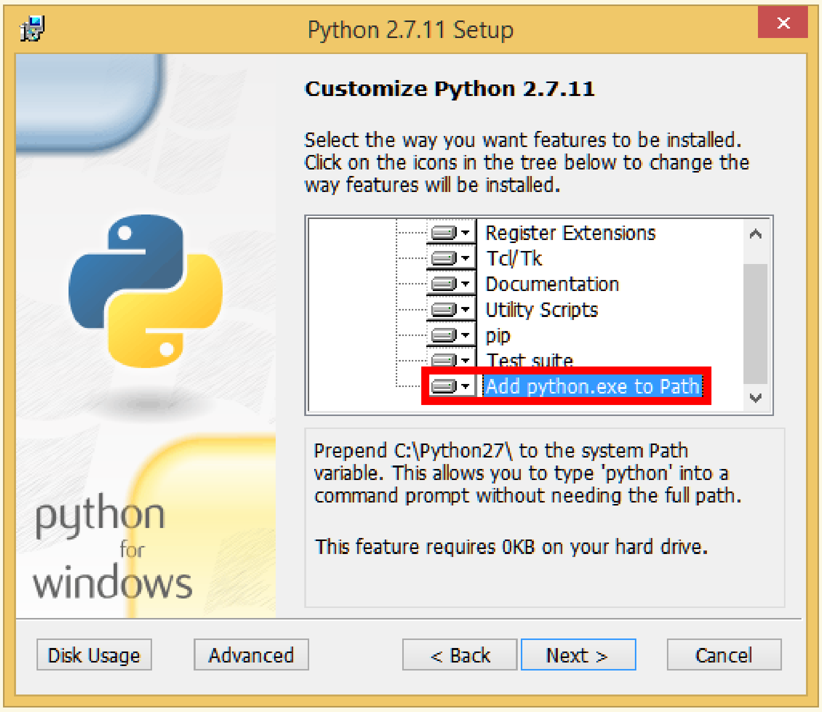
  OR

  A local installation of the Azure Cosmos DB Emulator.

- Visual Studio 2013 or higher, or , 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**.



Python 2.7.11 Setup

Customize Python 2.7.11

Select the way you want features to be installed. Click on the icons in the tree below to change the way features will be installed.

Prepend C:\Python27\ to the system Path variable. This allows you to type 'python' into a command prompt without needing the full path.

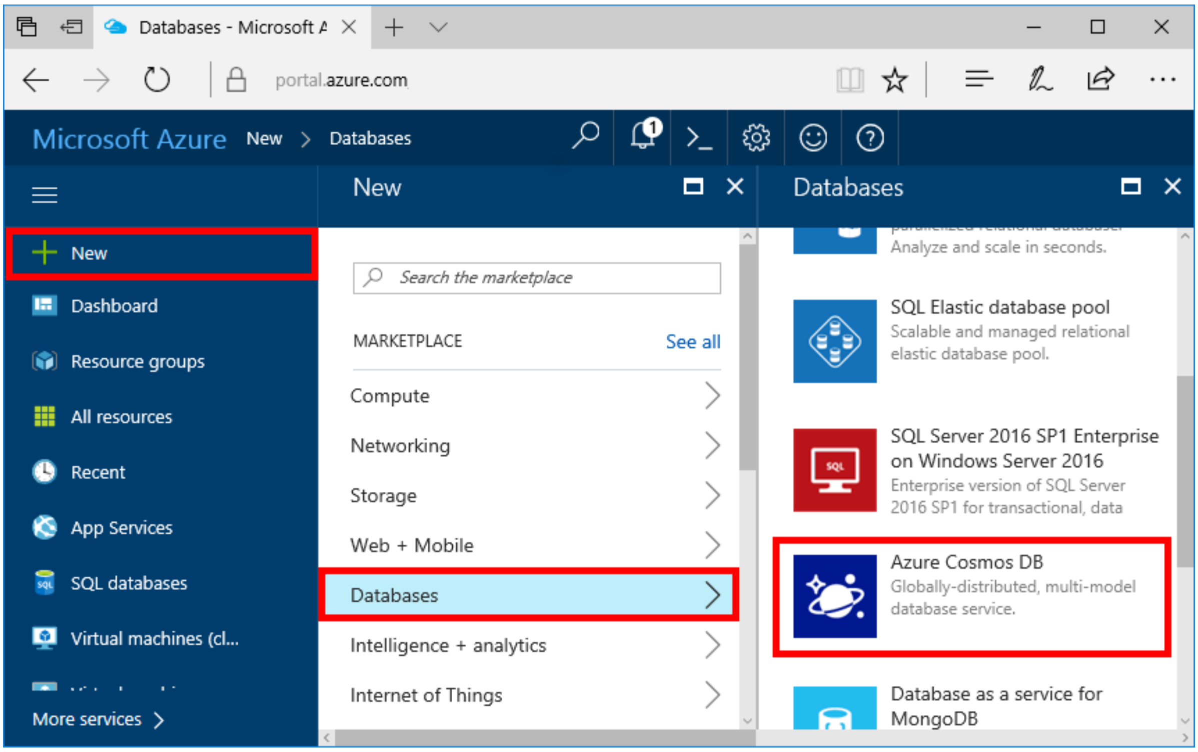
This feature requires 0KB on your hard drive.

- Microsoft Visual C++ Compiler for Python 2.7 from the Microsoft Download Center.

Step 1: Create an Azure Cosmos DB database account

Let's start by creating an Cosmos DB account. If you already have an account or if you are using the Azure Cosmos DB Emulator for this tutorial, you can skip to Step 2: Create a new Python Flask web application.

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

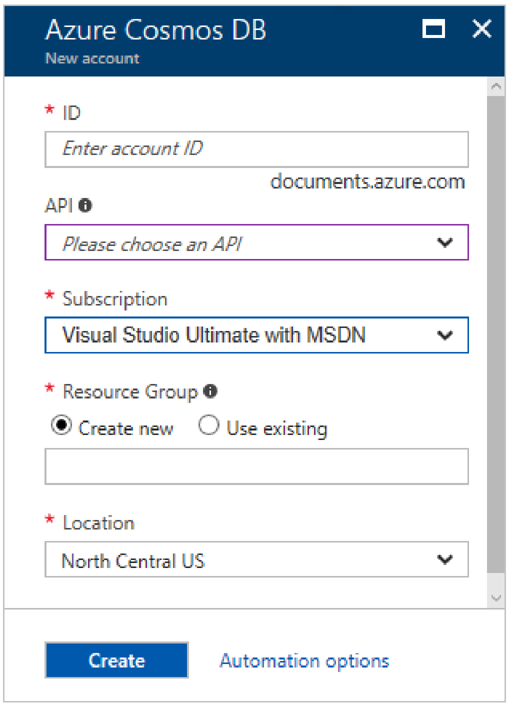


3. On the **New account** blade, specify the configuration that you want for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article we program against the DocumentDB API, so choose **SQL (DocumentDB)** as you fill out the form. But if you have graph data for a social media app, or key/value (table) data, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

   Complete the fields on the **New account** blade, using the information in the following screenshot as a guide. When you set up your account, choose unique values that do not match those in the screenshot.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

* Subscription
Visual Studio Ultimate with MSDN

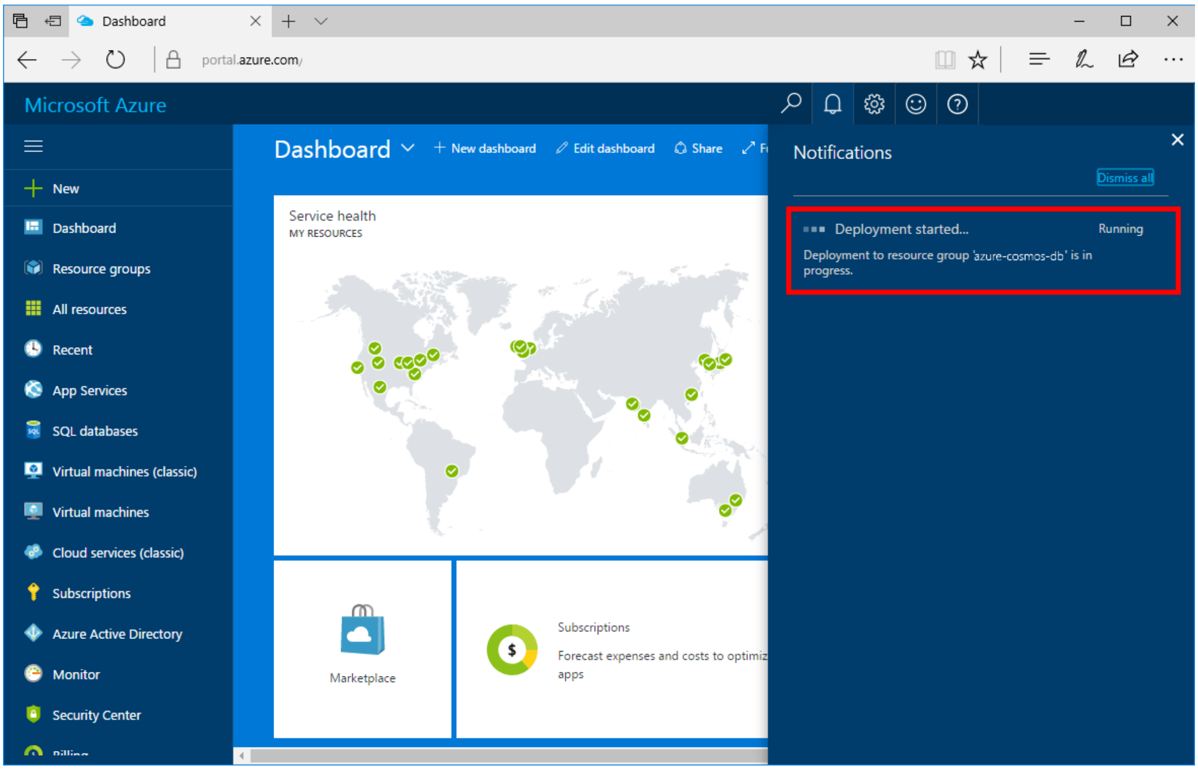
* Resource Group
● Create new ○ Use existing

* Location
North Central US

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that identifies your Azure Cosmos DB account. The string *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 through 50 characters.
API	SQL (DocumentDB)	We program against the DocumentDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for your Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource-group name for your account. For simplicity, you can use the same name as your ID.

SETTING	SUGGESTED VALUE	DESCRIPTION
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location that's closest to your users to give them the fastest access to the data.

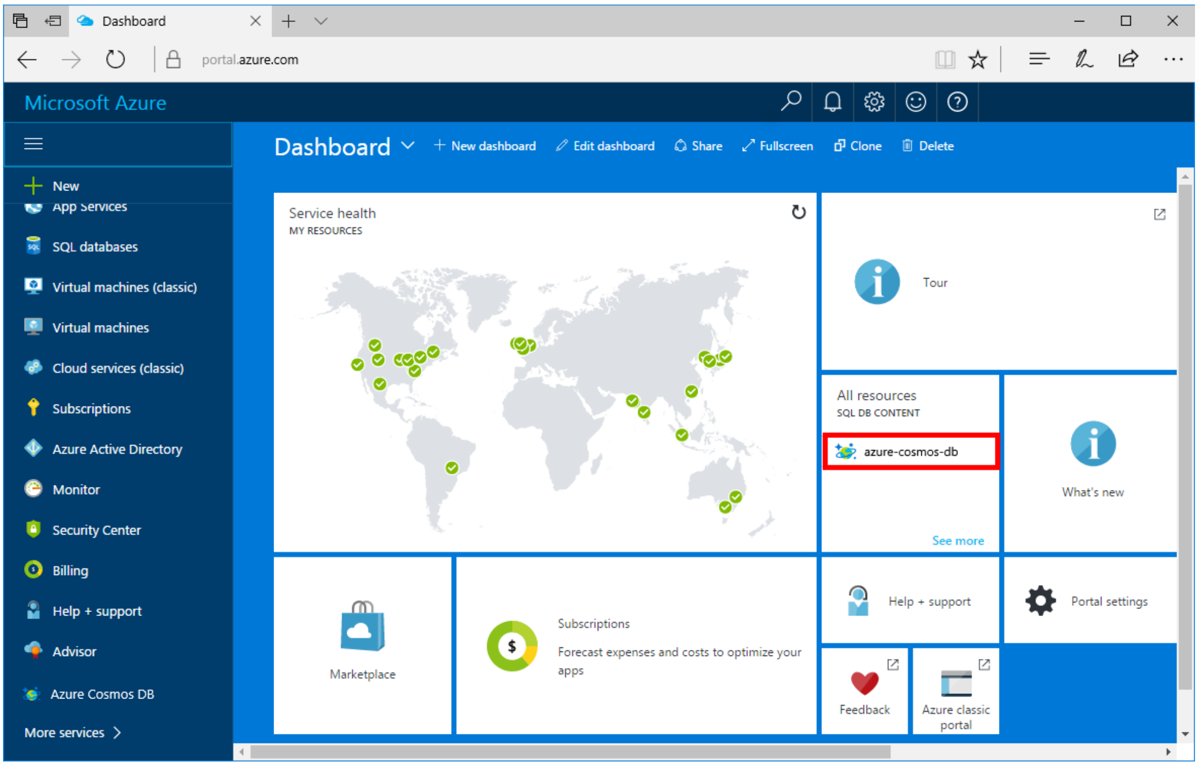
4. Click **Create** to create the account.
5. On the top toolbar, click **Notifications** to monitor the deployment process.



Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

6. When the deployment is complete, open the new account from the **All Resources** tile.



We will now walk through how to create a new Python Flask web application from the ground up.

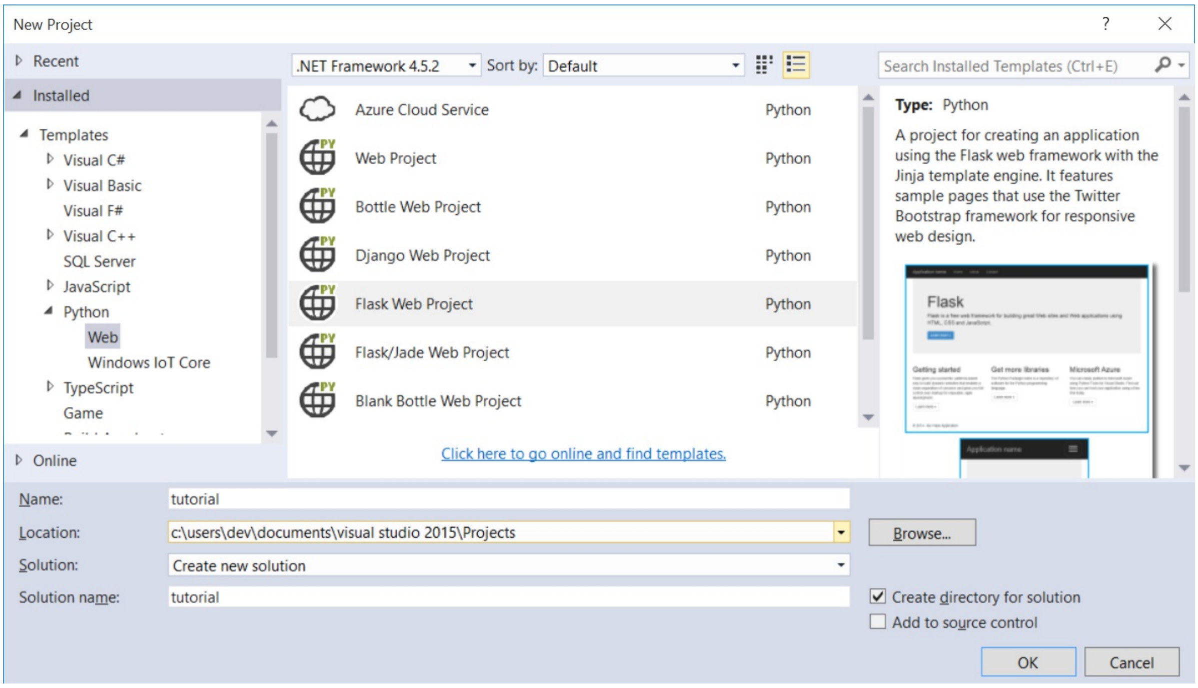
Step 2: Create a new Python Flask web application

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.



Type: Python

A project for creating an application using the Flask web framework with the Jinja template engine. It features sample pages that use the Twitter Bootstrap framework for responsive web design.

Name: tutorial

Location: c:\users\dev\documents\visual studio 2015\Projects

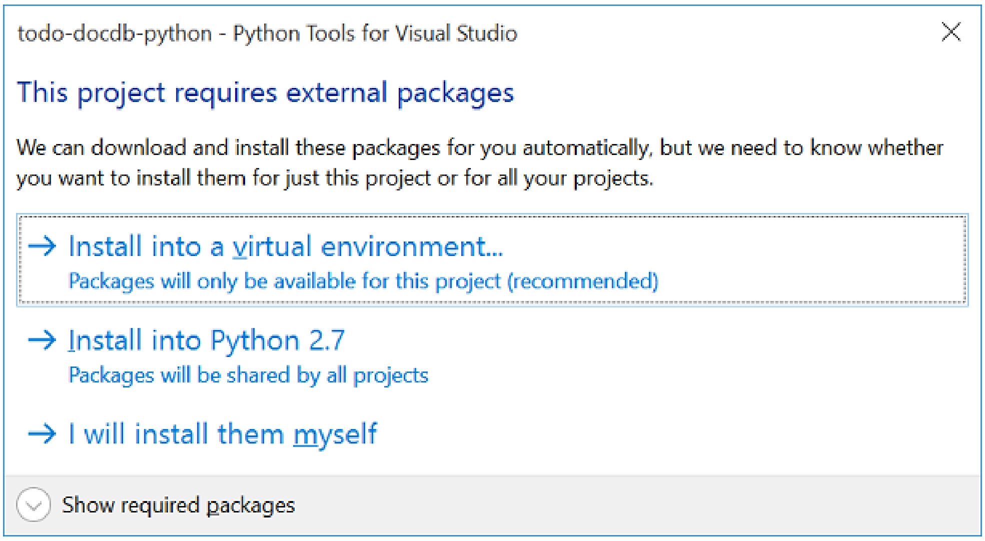
Solution: Create new solution

Solution name: tutorial

☑ Create directory for solution

☐ Add to source control

4. In the **Python Tools for Visual Studio** window, click **Install into a virtual environment**.



todo-docdb-python - Python Tools for Visual Studio

This project requires external packages

We can download and install these packages for you automatically, but we need to know whether you want to install them for just this project or for all your projects.

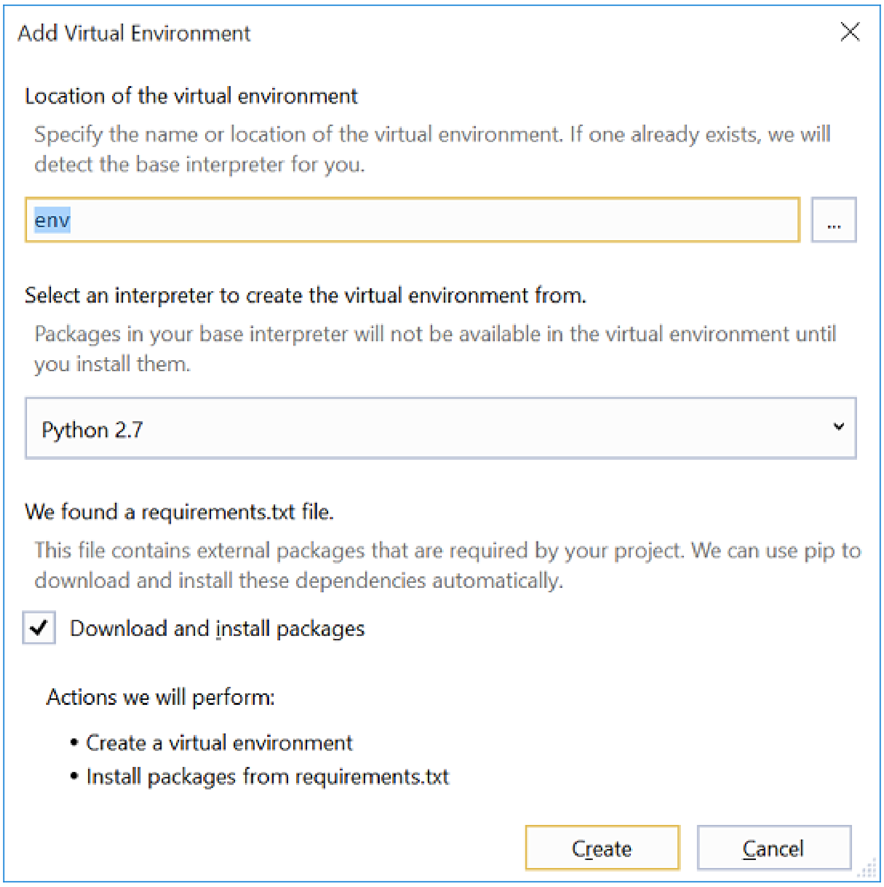
→ Install into a virtual environment...
Packages will only be available for this project (recommended)

→ Install into Python 2.7
Packages will be shared by all projects

→ I will install them myself

Show required packages

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.



Location of the virtual environment

Specify the name or location of the virtual environment. If one already exists, we will detect the base interpreter for you.

env

Select an interpreter to create the virtual environment from.

Packages in your base interpreter will not be available in the virtual environment until you install them.

Python 2.7

We found a requirements.txt file.

This file contains external packages that are required by your project. We can use pip to download and install these dependencies automatically.

☑ Download and install packages

Actions we will perform:

- Create a virtual environment
- Install packages from requirements.txt

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

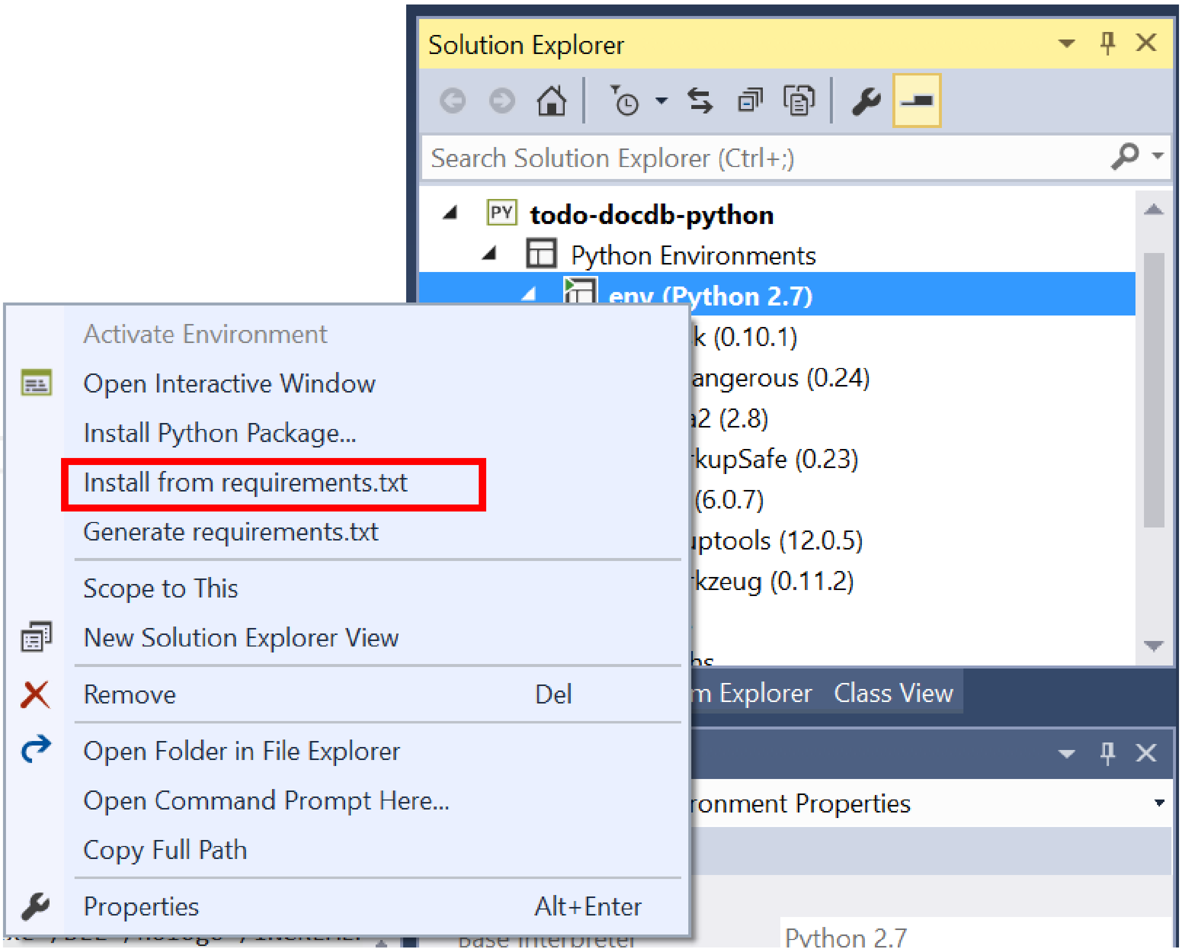
Add the Python Flask packages to your project

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:

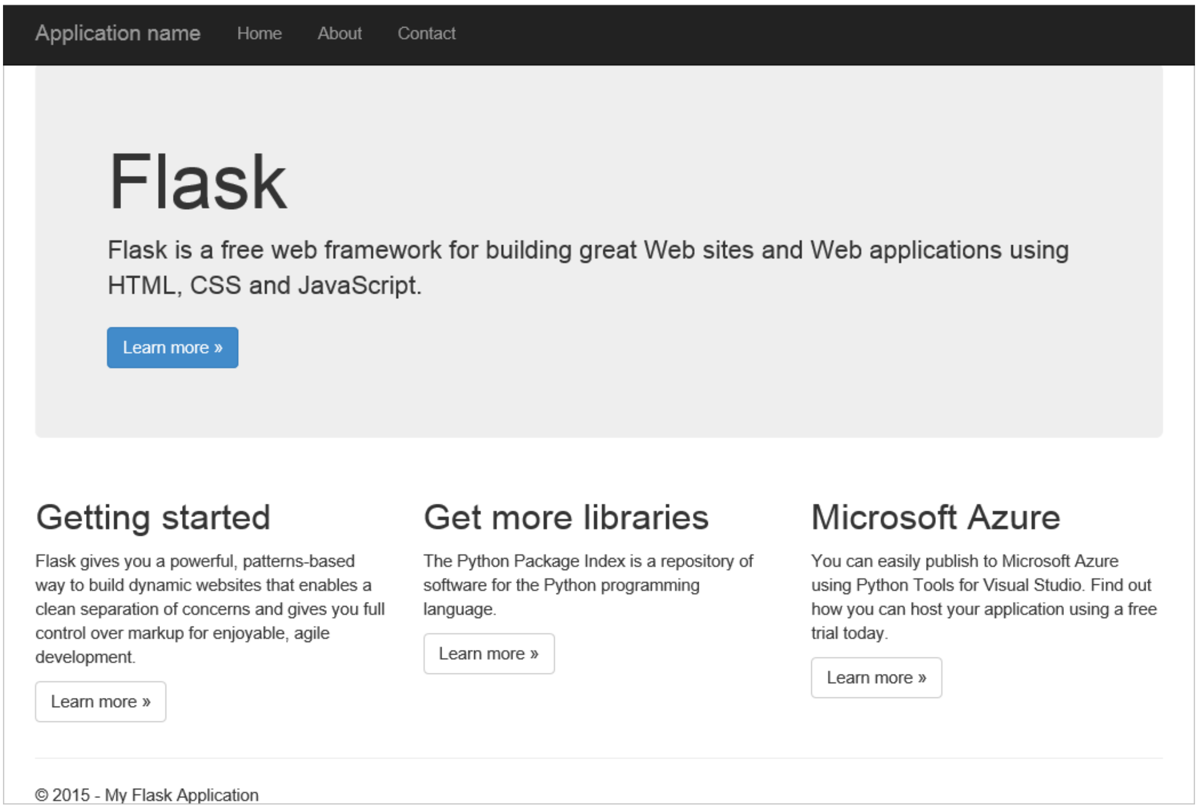
```
Successfully installed Babel-2.3.2 Tempita-0.5.2 WTForms-2.1 Whoosh-2.7.4 blinker-1.4 decorator-4.0.9 flask-0.9 flask-babel-0.8 flask-mail-0.7.6 flask-sqlalchemy-0.16 flask-whooshalchemy-0.55a0 flask-wtf-0.8.4 flup-1.0.2 pydocumentdb-1.6.1 pytz-2013b0 speaklater-1.3 sqlalchemy-0.7.9 sqlalchemy-migrate-0.7.2
```

> 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.



Flask

Flask is a free web framework for building great Web sites and Web applications using HTML, CSS and JavaScript.

Learn more »

Getting started

Flask gives you a powerful, patterns-based way to build dynamic websites that enables a clean separation of concerns and gives you full control over markup for enjoyable, agile development.

Learn more »

Get more libraries

The Python Package Index is a repository of software for the Python programming language.

Learn more »

Microsoft Azure

You can easily publish to Microsoft Azure using Python Tools for Visual Studio. Find out how you can host your application using a free trial today.

Learn more »

© 2015 - My Flask Application

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 Cosmos DB'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 Cosmos DB Offer Types, see Performance levels in Azure Cosmos DB.

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 + Azure Cosmos DB 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'
```

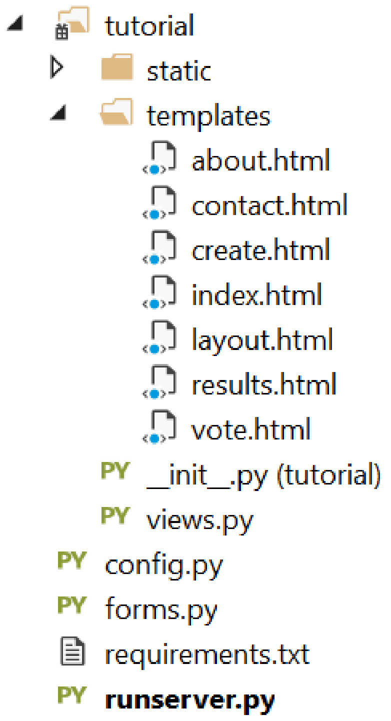
3. In the Azure portal, navigate to the **Keys** blade by clicking **Browse**, **Azure Cosmos DB Accounts**, double-click the name of the account to use, and then click the **Keys** button in the **Essentials** area. In the **Keys** blade, copy the **URI** value and paste it into the **config.py** file, 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
```

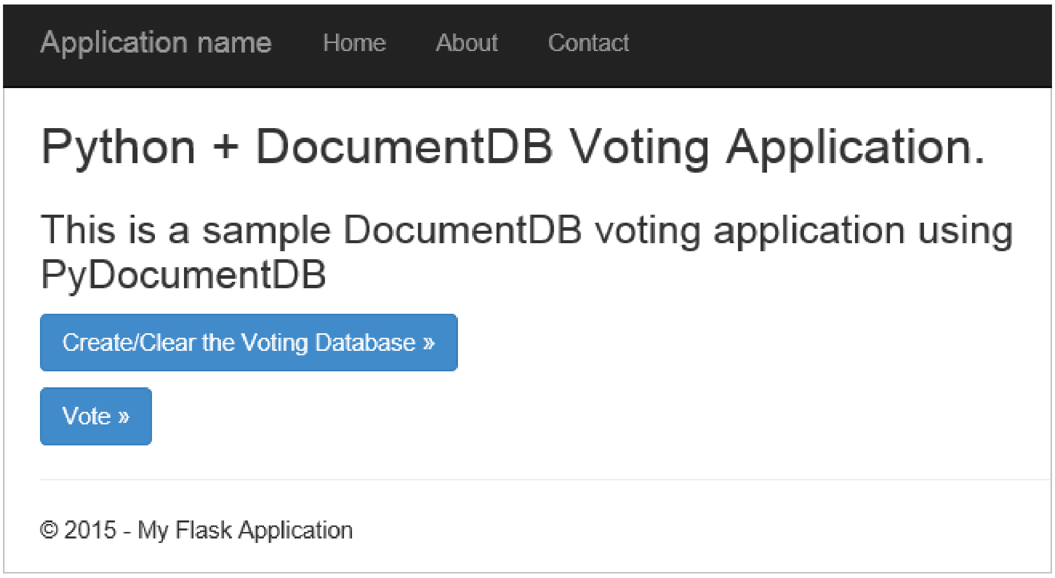
6. After adding all the files, Solution Explorer should look like this:



```
◢ tutorial
    ▷ static
    ◢ templates
        about.html
        contact.html
        create.html
        index.html
        layout.html
        results.html
        vote.html
      __init__.py (tutorial)
      views.py
  config.py
  forms.py
  requirements.txt
  runserver.py
```

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.

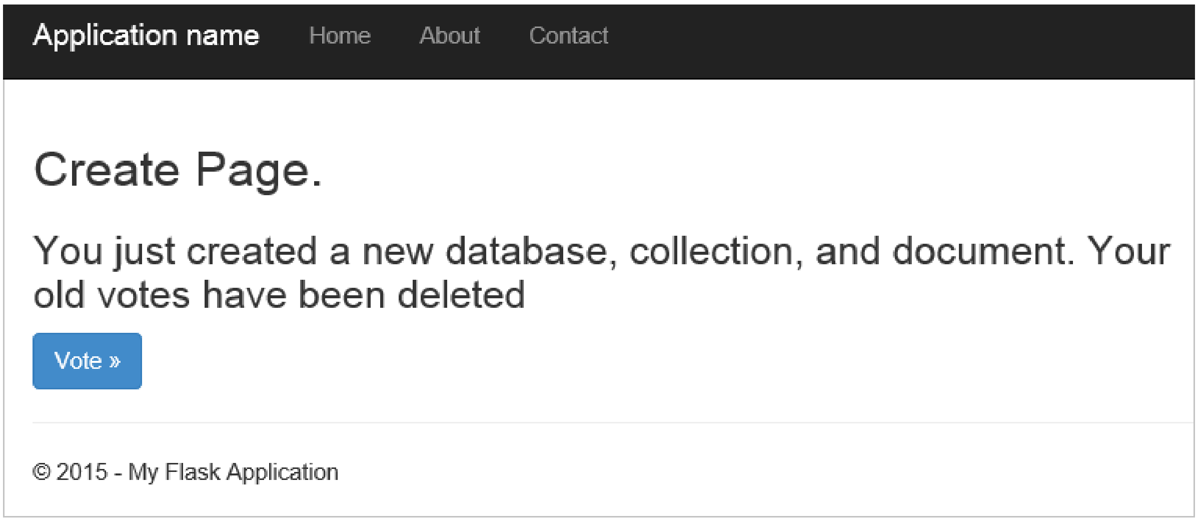


Python + DocumentDB Voting Application.

This is a sample DocumentDB voting application using PyDocumentDB

© 2015 - My Flask Application

3. Click **Create/Clear the Voting Database** to generate the database.

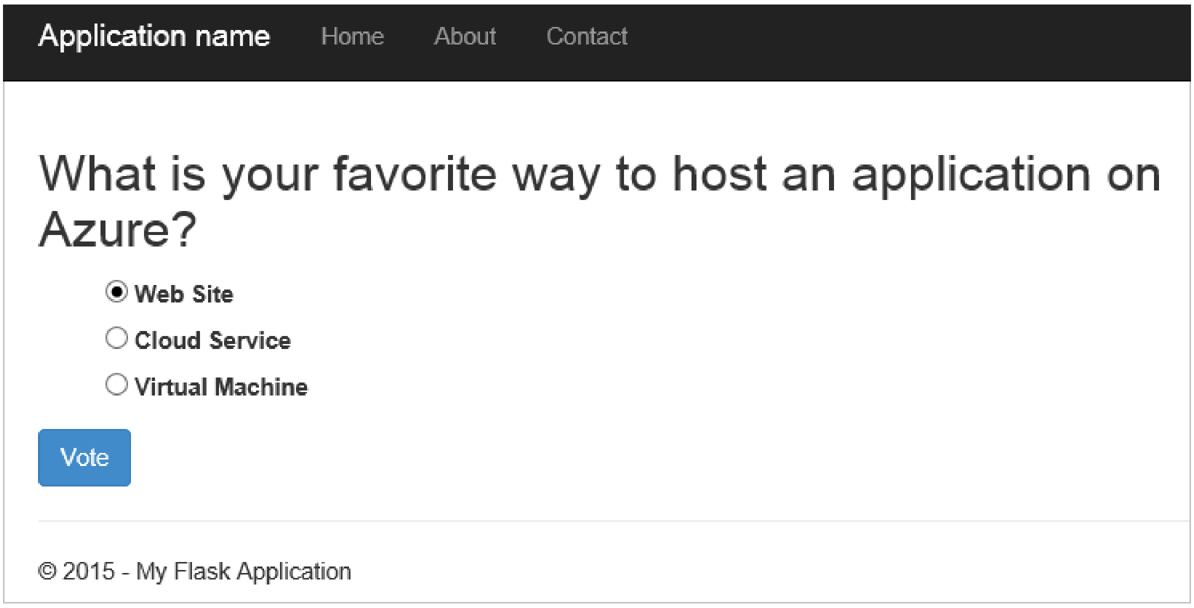


Create Page.

You just created a new database, collection, and document. Your old votes have been deleted

© 2015 - My Flask Application

4. Then, click **Vote** and select your option.

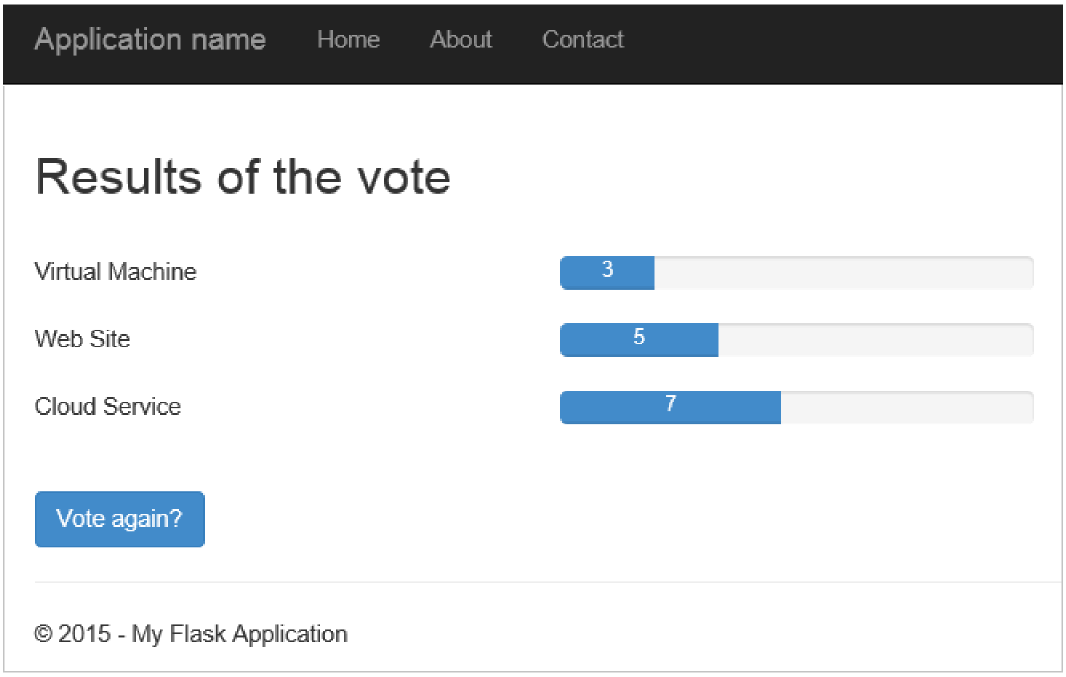


What is your favorite way to host an application on Azure?

● Web Site
○ Cloud Service
○ Virtual Machine

© 2015 - My Flask Application

5. For every vote you cast, it increments the appropriate counter.



Results of the vote

Virtual Machine	3
Web Site	5
Cloud Service	7

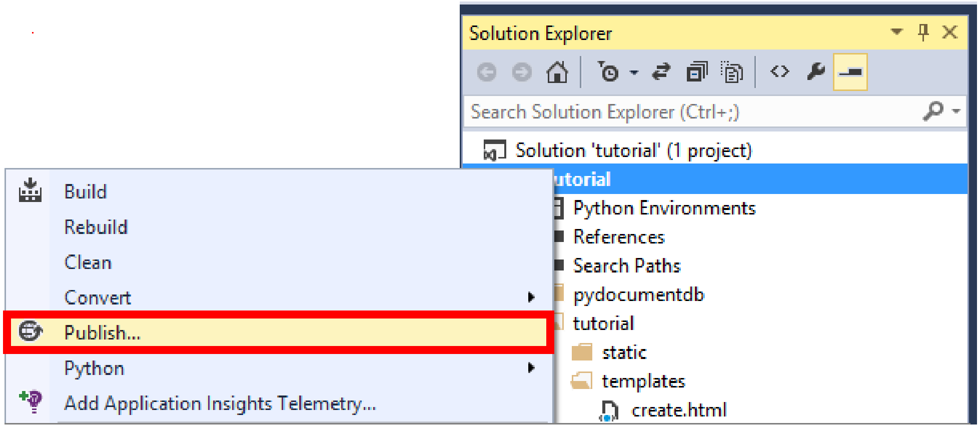
© 2015 - My Flask Application

6. Stop debugging the project by pressing Shift+F5.

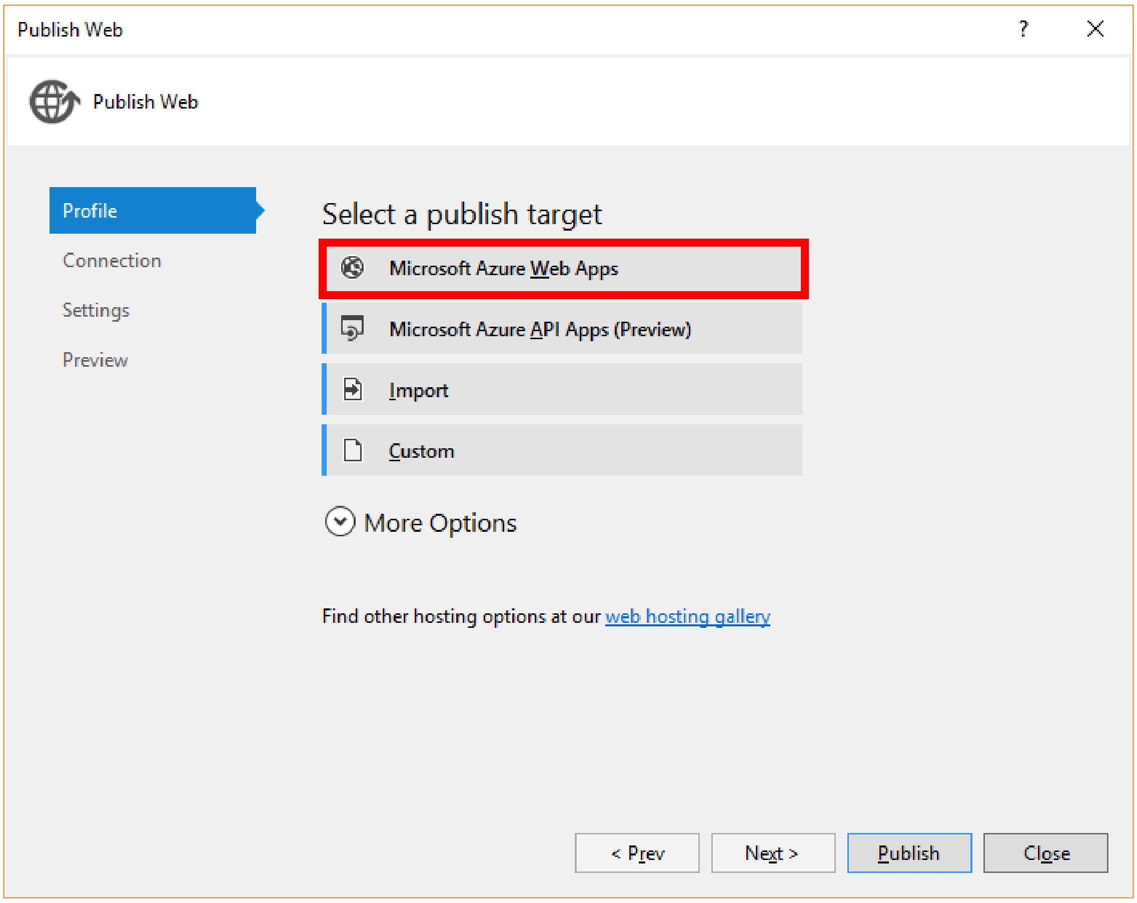
Step 5: Deploy the web application to Azure Websites

Now that you have the complete application working correctly against Cosmos DB, 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**.



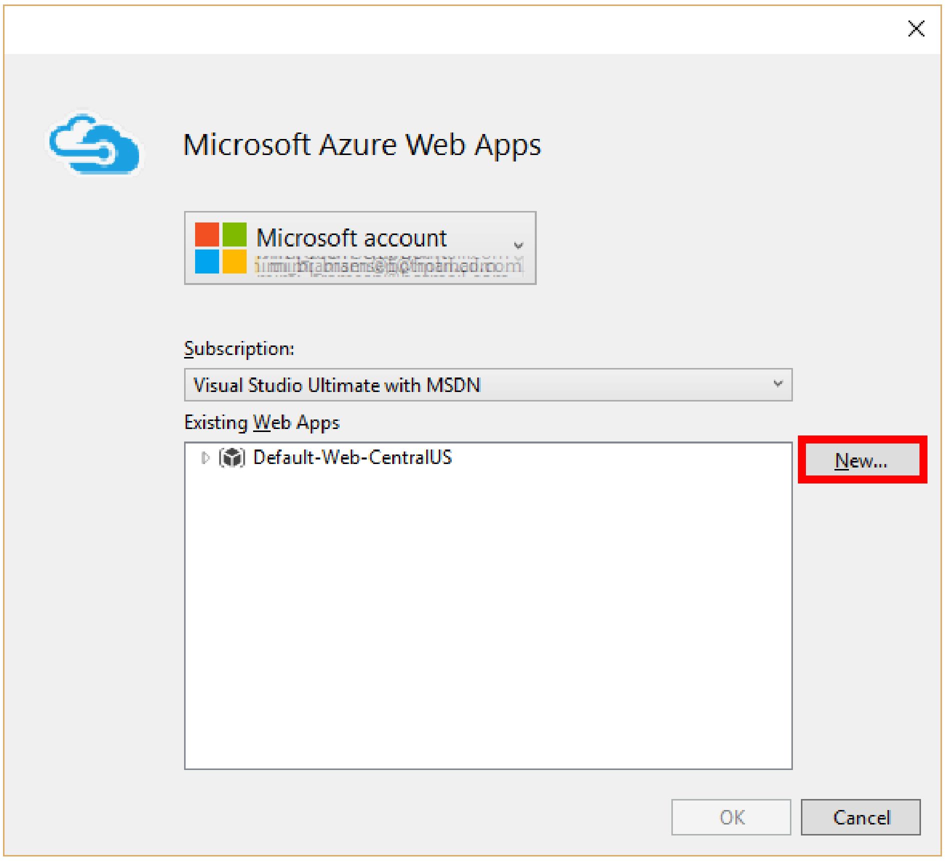
Publish Web

Publish Web

Select a publish target

Find other hosting options at our web hosting gallery

3. In the **Microsoft Azure Web Apps Window** window, click **New**.



Microsoft Azure Web Apps

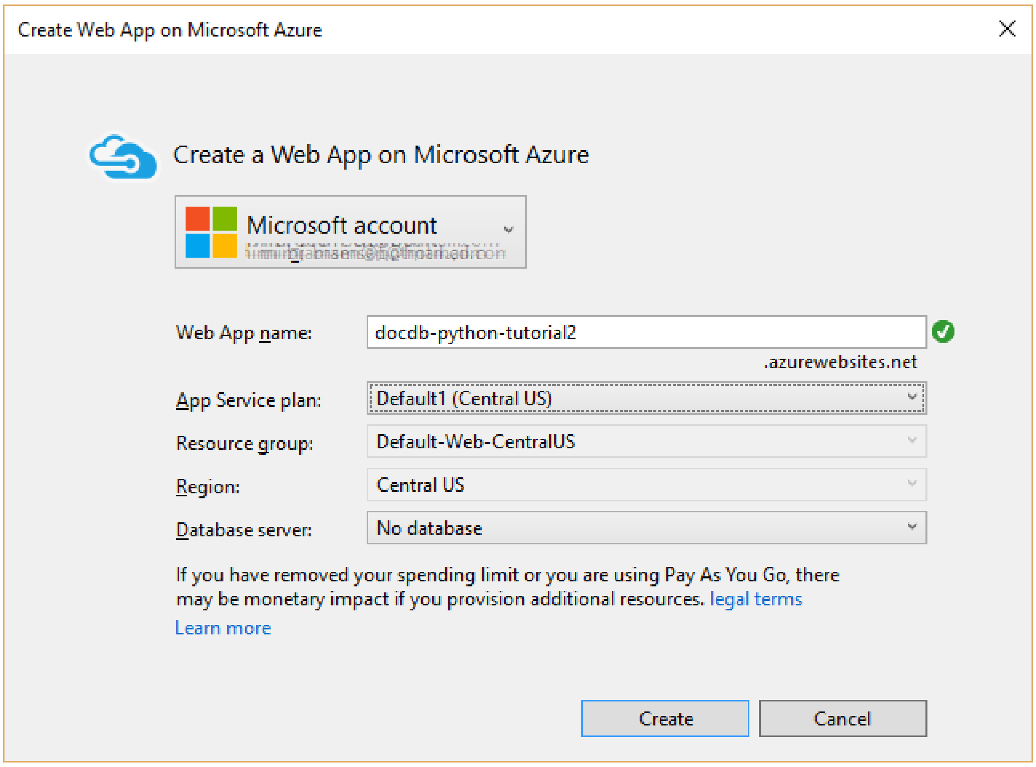
Microsoft account
[illegible]

Subscription: Visual Studio Ultimate with MSDN

Existing Web Apps

Default-Web-CentralUS

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

[illegible]

Web App name: docdb-python-tutorial2 .azurewebsites.net

App Service plan: Default1 (Central US)

Resource group: Default-Web-CentralUS

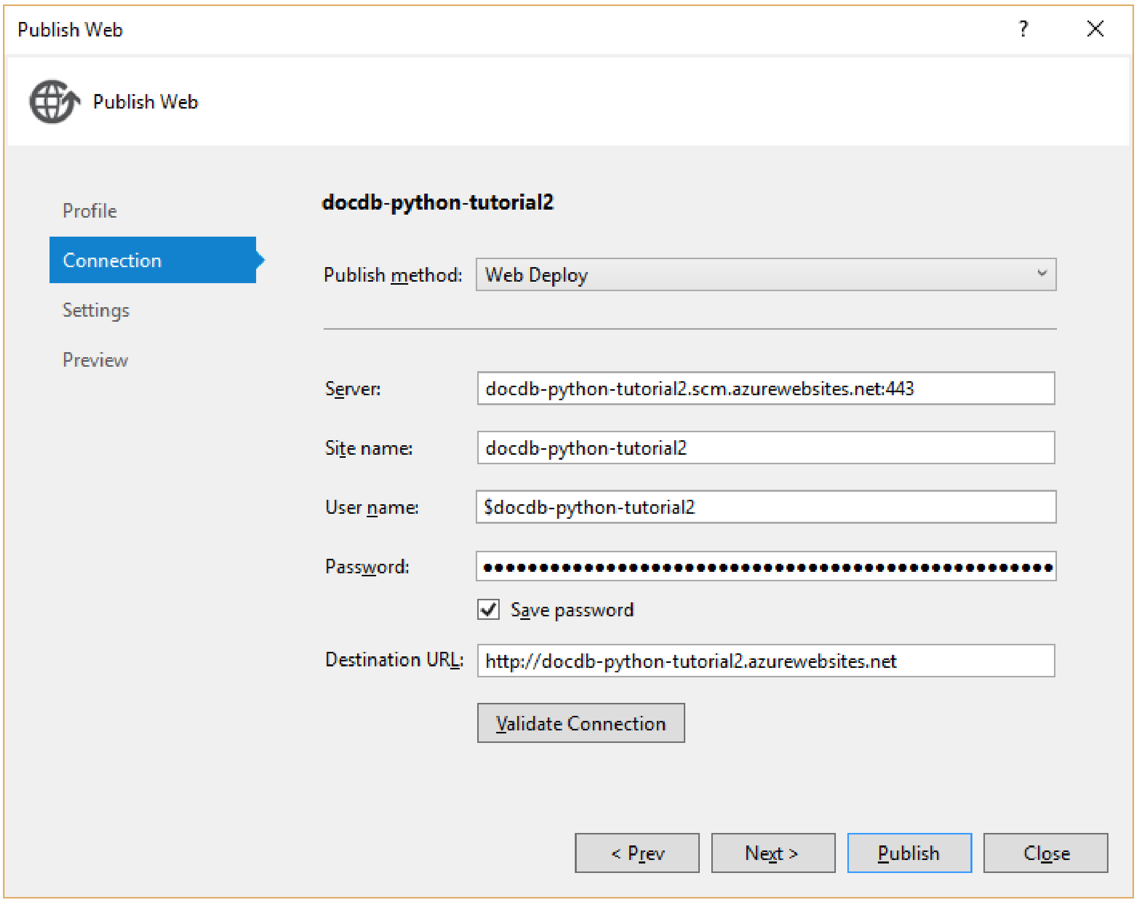
Region: Central US

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

5. In the **Publish Web** window, click **Publish**.



Publish Web

Publish Web

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

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 Cosmos DB 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!.

DocumentDB .NET examples

5/30/2017 • 3 min to read • Edit Online

Latest sample solutions that perform CRUD operations and other common operations on Azure Cosmos DB resources are included in the azure-documentdb-dotnet 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 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 a database	DocumentQueryable.CreateDatabaseQuery
Read a database by Id	DocumentClient.ReadDatabaseAsync
Read all the databases	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 configured performance of a collection	DocumentQueryable.CreateOfferQuery
Change configured performance of a collection	DocumentClient.ReplaceOfferAsync
Get a collection by Id	DocumentClient.ReadDocumentCollectionAsync
Read all the 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 the 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

TASK	API REFERENCE
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.

Geospatial examples

The geospatial sample file, azure-documentdb-dotnet/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 [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 [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) 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 Cosmos DB.

Query examples

The query document file, azure-documentdb-dotnet/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
Query using range operators against strings	DocumentQueryable.CreateDocumentQuery
Query with Order by	DocumentQueryable.CreateDocumentQuery
Query with Aggregate Functions	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 explict 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-dotnet/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

TASK	API REFERENCE
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 Azure Cosmos DB server-side programming: Stored procedures, database triggers, and UDFs.

User management examples

The user management file, azure-documentdb-dotnet/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

Azure Cosmos DB Node.js examples

5/30/2017 • 2 min to read • Edit Online

Sample solutions that perform CRUD operations and other common operations on Azure Cosmos DB 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

TASK	API REFERENCE
Read a list of all collections in a database	DocumentClient.readCollections
Get a collection by _self	DocumentClient.readCollection
Get a collection by Id	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.createCollection
Manually index a specific document	RequestOptions.indexingDirective: 'include'
Manually exclude a specific document from the index	RequestOptions.indexingDirective: 'exclude'
Use lazy indexing for bulk import or read heavy collections	IndexingMode.Lazy

TASK	API REFERENCE
Include specific paths of a document in indexing	IndexingPolicy.IncludedPaths
Exclude certain paths from indexing	IndexingPolicy.ExcludedPath
Allow a scan on a string path during a range operation	FeedOptions.EnableScanInQuery
Create a range index on a string path	IndexKind.Range, IndexingPolicy, DocumentClient.queryDocument
Create a collection with default indexPolicy, then update this online	DocumentClient.createCollection DocumentClient.replaceCollection#replaceCollection

For more information about indexing, see Azure Cosmos DB 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 Azure Cosmos DB 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 Azure Cosmos DB, see Partition and scale data in Azure Cosmos DB.

Azure Cosmos DB Python examples

5/30/2017 • 1 min to read • Edit Online

Sample solutions that perform CRUD operations and other common operations on Azure Cosmos DB 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	document_client.CreateDatabase
Query an account for a database	document_client.QueryDatabases
Read a database by Id	document_client.ReadDatabase
List databases for an account	document_client.ReadDatabases
Delete a database	document_client.DeleteDatabase

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

TASK	API REFERENCE
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

Azure Cosmos DB: DocumentDB API SQL query cheat sheet PDF

6/9/2017 • 1 min to read • Edit Online

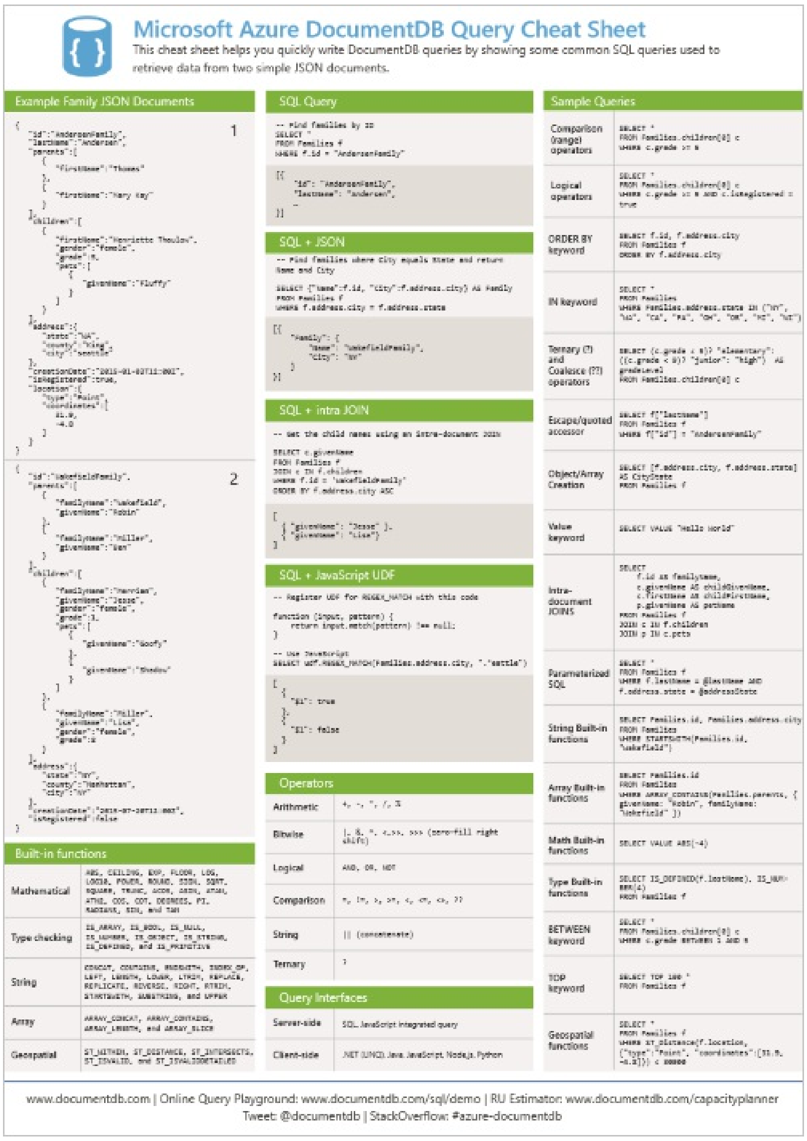
The **Azure Cosmos DB: DocumentDB API SQL Query Cheat Sheet** helps you quickly write queries for DocumentDB API data by displaying common database queries, keywords, built-in functions, and operators in an easy to print PDF reference sheet.

Cosmos DB 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, Cosmos DB supports JavaScript user defined functions (UDFs), JavaScript operators, and a multitude of built-in functions.

Download the Cosmos DB 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 Cosmos DB 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

1

```
{
   "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":"2015-01-03T12:00Z",
   "isRegistered":true,
   "location":{
      "type":"Point",
      "coordinates":[
         31.9,
         -4.8
      ]
   }
}
```

2

```
{
   "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":"2015-07-20T12:00Z",
   "isRegistered":false
}
```

Built-in functions

Mathematical	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	IS_ARRAY, IS_BOOL, IS_NULL, IS_NUMBER, IS_OBJECT, IS_STRING, IS_DEFINED, and IS_PRIMITIVE
String	CONCAT, CONTAINS, ENDSWITH, INDEX_OF, LEFT, LENGTH, LOWER, LTRIM, REPLACE, REPLICATE, REVERSE, RIGHT, RTRIM, STARTSWITH, SUBSTRING, and UPPER
Array	ARRAY_CONCAT, ARRAY_CONTAINS, ARRAY_LENGTH, and ARRAY_SLICE
Geospatial	ST_WITHIN, ST_DISTANCE, ST_INTERSECTS, ST_ISVALID, and ST_ISVALIDDETAILED

SQL Query

```
-- Find families by ID
SELECT *
FROM Families f
WHERE f.id = "AndersenFamily"
```

```
[{
    "id": "AndersenFamily",
    "lastName": "Andersen",
    …
}]
```

SQL + JSON

```
-- Find families where city equals State and return
Name and City

SELECT {"Name":f.id, "City":f.address.city} AS Family
FROM Families f
WHERE f.address.city = f.address.state
```

```
[{
   "Family": {
       "Name": "WakefieldFamily",
       "City": "NY"
   }
}]
```

SQL + intra JOIN

```
-- Get the child names using an intra-document JOIN

SELECT c.givenName
FROM Families f
JOIN c IN f.children
WHERE f.id = 'WakefieldFamily'
ORDER BY f.address.city ASC
```

```
[
  { "givenName": "Jesse" },
  { "givenName": "Lisa"}
]
```

SQL + JavaScript UDF

```
-- Register UDF for REGEX_MATCH with this code

function (input, pattern) {
    return input.match(pattern) !== null;
}

-- Use JavaScript
SELECT udf.REGEX_MATCH(Families.address.city, ".*eattle")
```

```
[
  {
    "$1": true
  },
  {
    "$1": false
  }
]
```

Operators

Arithmetic	+, -, *, /, %
Bitwise	|, &, ^, <<, >>, >>> (zero-fill right shift)
Logical	AND, OR, NOT
Comparison	=, !=, >, >=, <, <=, <>, ??
String	|| (concatenate)
Ternary	?

Query Interfaces

Server-side	SQL, JavaScript integrated query
Client-side	.NET (LINQ), Java, JavaScript, Node.js, Python

Sample Queries

Comparison (range) operators	SELECT *
	FROM Families.children[0] c
	WHERE c.grade >= 5

Logical operators	SELECT *
	FROM Families.children[0] c
	WHERE c.grade >= 5 AND c.isRegistered = true

ORDER BY keyword	SELECT f.id, f.address.city
	FROM Families f
	ORDER BY f.address.city

IN keyword	SELECT *
	FROM Families
	WHERE Families.address.state IN ("NY", "WA", "CA", "PA", "OH", "OR", "MI", "WI")

Ternary (?) and Coalesce (??) operators	SELECT (c.grade < 5)? "elementary": ((c.grade < 9)? "junior": "high") AS gradeLevel
	FROM Families.children[0] c

Escape/quoted accessor	SELECT f["lastName"]
	FROM Families f
	WHERE f["id"] = "AndersenFamily"

Object/Array Creation	SELECT [f.address.city, f.address.state] AS CityState
	FROM Families f

Value keyword	SELECT VALUE "Hello World"
Intra-document JOINS	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

Parameterized SQL	SELECT *
	FROM Families f
	WHERE f.lastName = @lastName AND f.address.state = @addressState

String Built-in functions	SELECT Families.id, Families.address.city
	FROM Families
	WHERE STARTSWITH(Families.id, "Wakefield")

Array Built-in functions	SELECT Families.id
	FROM Families
	WHERE ARRAY_CONTAINS(Families.parents, { givenName: "Robin", familyName: "Wakefield" })

Math Built-in functions	SELECT VALUE ABS(-4)

Type Built-in functions	SELECT IS_DEFINED(f.lastName), IS_NUMBER(4)
	FROM Families f

BETWEEN keyword	SELECT *
	FROM Families.children[0] c
	WHERE c.grade BETWEEN 1 AND 5

TOP keyword	SELECT TOP 100 *
	FROM Families f

Geospatial functions	SELECT *
	FROM Families f
	WHERE ST_DISTANCE(f.location, {"type":"Point", "coordinates":[31.9, -4.8]}) < 30000

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 Cosmos DB, see Query Cosmos DB.
- For the related reference documentation, see Cosmos DB SQL Query Language.

Release notes

Updated on 7/29/2016 to include TOP.

Community portal

5/30/2017 • 10 min to read • Edit Online

Community spotlight

Let us promote your project! Show us the awesome project you're working on with Azure Cosmos DB, the next generation of the DocumentDB stack, and we will help share your genius with the world. To submit your project, send us an e-mail at: askcosmosdb@microsoft.com.

documentdb-lumenize

by Larry Maccherone

Aggregations (Group-by, Pivot-table, and N-dimensional Cube) and Time Series Transformations as Stored Procedures in Azure Cosmos DB DocumentDB API.

Check it out on GitHub and npm.

DocumentDB Studio

by Ming Liu

A client management viewer/explorer for the Azure Cosmos DB DocumentDB API 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 Azure Cosmos DB.

Check it out on GitHub and npm.

TypeScript API

by Jelmer Cormont

A wrapper around the Node.js client written in TypeScript (works in plain JavaScript too). Supports `async/await` and a simplified API.

Check it out on GitHub and npm.

Swagger REST API for DocumentDB

by Howard Edidin

An Azure Cosmos DB 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 Cosmos DB DocumentDB API.

Check it out on GitHub and rubygems.

Find more open source Azure Cosmos DB projects on GitHub.

News, blogs, and articles

You can stay up-to-date with the latest Azure Cosmos DB 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*
- **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 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*
- **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
South Florida Codecamp: NoSQL for .NET developers in under 10 minutes with Azure DocumentDB	Santosh Hari	Davie, FL	March 11, 2017	#sflcc
Orlando Codecamp: NoSQL for .NET developers in under 10 minutes with Azure DocumentDB	Santosh Hari	Sanford, FL	April 8, 2017	#OrlandoCC

EVENT NAME	SPEAKER	LOCATION	DATE	HASHTAG
Global Azure Bootcamp: Serverless computing in Azure with Azure Functions and DocumentDB	Josh Lane	Atlanta, GA	April 22, 2017	#GlobalAzure
NDC Olso 2017: Azure DocumentDB - The Best NoSQL Database You're Probably Not Using (Yet)	Josh Lane	Olso, Norway	June 14, 2017	#ndcoslo

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
Ignite Australia: Hello DocumentDB: Azure's blazing fast, planet-scale NoSQL database	Andrew Liu	Queensland, Australia	Wednesday February 15, 2017	Forthcoming
Ignite Australia: A Deep-Dive with Azure DocumentDB: Partitioning, Data Modelling, and Geo Replication	Andrew Liu	Queensland, Australia	February 16, 2017	Forthcoming
Wintellect webinar: An Introduction to Azure DocumentDB	Josh Lane	Online	January 12, 2017 1pm EST	Azure DocumentDB: Your Cloud-powered, Geo-scaled, NoSQL Superweapon... Hiding in Plain Sight
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
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

EVENT NAME	SPEAKER	LOCATION	DATE	RECORDING
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
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

EVENT NAME	SPEAKER	LOCATION	DATE	RECORDING
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
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

EVENT NAME	SPEAKER	LOCATION	DATE	RECORDING
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
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

SHOW	SPEAKER	DATE	EPISODE
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	October 5, 2015	Azure DocumentDB for Healthcare Integration
DX Italy - #TecHeroes	Alessandro Melchiori	October 2, 2015	#TecHeroes - 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
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 Availibility and What's New!
Data Exposed	Andrew Liu	March 17, 2015	Java SDK for DocumentDB

SHOW	SPEAKER	DATE	EPISODE
#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
MVA	**Microsoft Virtual Academy** offers you training from the people who help build Azure DocumentDB.
pluralsight	**Pluralsight** is a key Microsoft partner offering Azure training. If you are an MSDN subscriber, use your benefits to access Microsoft Azure training.
ps	**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
stack overflow	A language-independent collaboratively edited question and answer site for programmers. Follow our tag: azure-documentdb

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 by sending us an e-mail or tweeting 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
Hadoop Connector	azure-documentdb-hadoop	Maven
Data migration tool	azure-documentdb-datamigrationtool	Microsoft download center

Azure Cosmos DB Wizards

Azure Cosmos DB Wizards are community leaders who’ve demonstrated an exemplary commitment to helping others get the most out of their experience with Azure Cosmos DB, the next generation of Azure DocumentDB. They share their exceptional passion, real-world knowledge, and technical expertise with the community and with the Azure Cosmos DB team.

WIZARD	PICTURE

WIZARD	PICTURE
Allan Mitchell	
Jen Stirrup	
Lenni Lobel	
Mihail Mateev	
Larry Maccherone	

WIZARD	PICTURE
Howard Edidin	
Santosh Hari	
Matías Quaranta	

Want to become an Azure Cosmos DB Wizard? While there is no benchmark for becoming a 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.

Retiring the S1, S2, and S3 performance levels

5/30/2017 • 7 min to read • Edit Online

IMPORTANT

The S1, S2, and S3 performance levels discussed in this article are being retired and are no longer available for new DocumentDB API accounts.

This article provides an overview of S1, S2, and S3 performance levels, and discusses how the collections that use these performance levels will be migrated to single partition collections on August 1st, 2017. After reading this article, you'll be able to answer the following questions:

- Why are the S1, S2, and S3 performance levels being retired?
- How do single partition collections and partitioned collections compare to the S1, S2, S3 performance levels?
- What do I need to do to ensure uninterrupted access to my data?
- How will my collection change after the migration?
- How will my billing change after I'm migrated to single partition collections?
- What if I need more than 10 GB of storage?
- Can I change between the S1, S2, and S3 performance levels before August 1, 2017?
- How will I know when my collection has migrated?
- How do I migrate from the S1, S2, S3 performance levels to single partition collections on my own?
- How am I impacted if I'm an EA customer?

Why are the S1, S2, and S3 performance levels being retired?

The S1, S2, and S3 performance levels do not offer the flexibility that DocumentDB API collections offers. With the S1, S2, S3 performance levels, both the throughput and storage capacity were pre-set and did not offer elasticity. Azure Cosmos DB now offers the ability to customize your throughput and storage, offering you much more flexibility in your ability to scale as your needs change.

How do single partition collections and partitioned collections compare to the S1, S2, S3 performance levels?

The following table compares the throughput and storage options available in single partition collections, partitioned collections, and S1, S2, S3 performance levels. Here is an example for US East 2 region:

	PARTITIONED COLLECTION	SINGLE PARTITION COLLECTION	S1	S2	S3
Maximum throughput	Unlimited	10K RU/s	250 RU/s	1 K RU/s	2.5 K RU/s
Minimum throughput	2.5K RU/s	400 RU/s	250 RU/s	1 K RU/s	2.5 K RU/s
Maximum storage	Unlimited	10 GB	10 GB	10 GB	10 GB

	PARTITIONED COLLECTION	SINGLE PARTITION COLLECTION	S1	S2	S3
Price	Throughput: $6 / 100 RU/s Storage: $0.25/GB	Throughput: $6 / 100 RU/s Storage: $0.25/GB	$25 USD	$50 USD	$100 USD

Are you an EA customer? If so, see How am I impacted if I'm an EA customer?

What do I need to do to ensure uninterrupted access to my data?

Nothing, Cosmos DB handles the migration for you. If you have an S1, S2, or S3 collection, your current collection will be migrated to a single partition collection on July 31, 2017.

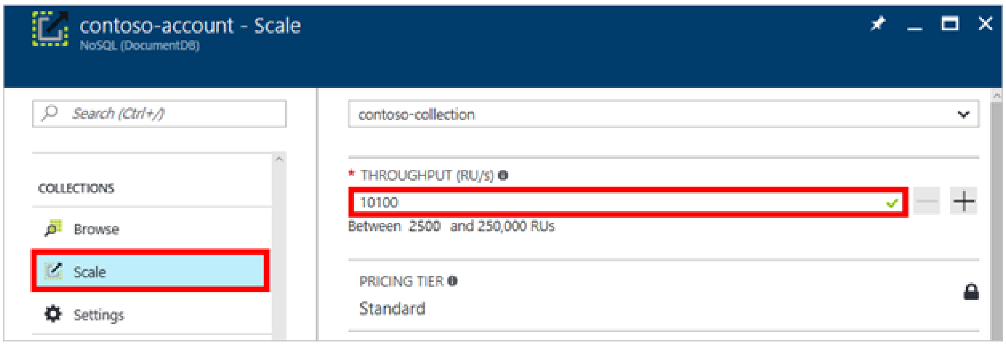
How will my collection change after the migration?

If you have an S1 collection, you will be migrated to a single partition collection with 400 RU/s throughput. 400 RU/s is the lowest throughput available with single partition collections. However, the cost for 400 RU/s in the a single partition collection is approximately the same as you were paying with your S1 collection and 250 RU/s – so you are not paying for the extra 150 RU/s available to you.

If you have an S2 collection, you will be migrated to a single partition collection with 1 K RU/s. You will see no change to your throughput level.

If you have an S3 collection, you will be migrated to a single partition collection with 2.5 K RU/s. You will see no change to your throughput level.

In each of these cases, after your collection is migrated, you will be able to customize your throughput level, or scale it up and down as needed to provide low-latency access to your users. To change the throughput level after your collection has migrated, simply open your Cosmos DB account in the Azure portal, click Scale, choose your collection, and then adjust the throughput level, as shown in the following screenshot:



contoso-account - Scale
NoSQL (DocumentDB)

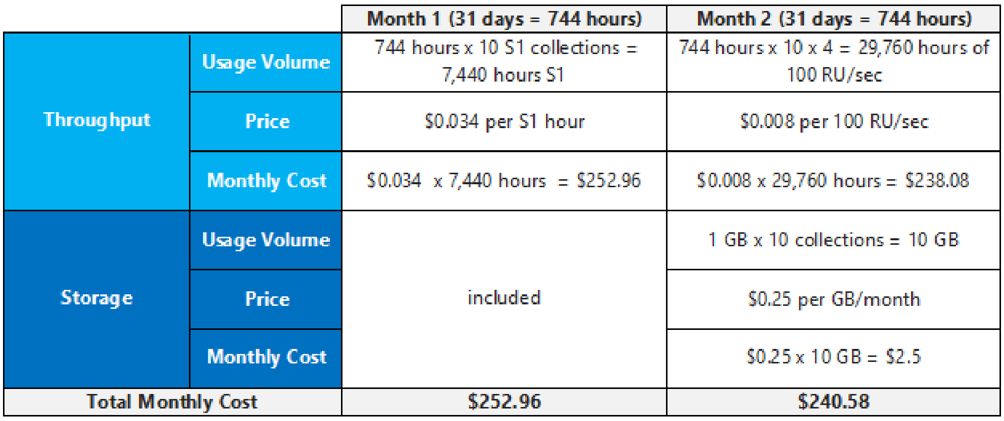
contoso-collection

* THROUGHPUT (RU/s): 10100
Between 2500 and 250,000 RUs

PRICING TIER: Standard

How will my billing change after I’m migrated to the single partition collections?

Assuming you have 10 S1 collections, 1 GB of storage for each, in the US East region, and you migrate these 10 S1 collections to 10 single partition collections at 400 RU/sec (the minimum level). Your bill will look as follows if you keep the 10 single partition collections for a full month:



		Month 1 (31 days = 744 hours)	Month 2 (31 days = 744 hours)
Throughput	Usage Volume	744 hours x 10 S1 collections = 7,440 hours S1	744 hours x 10 x 4 = 29,760 hours of 100 RU/sec
	Price	$0.034 per S1 hour	$0.008 per 100 RU/sec
	Monthly Cost	$0.034 x 7,440 hours = $252.96	$0.008 x 29,760 hours = $238.08
Storage	Usage Volume	included	1 GB x 10 collections = 10 GB
	Price		$0.25 per GB/month
	Monthly Cost		$0.25 x 10 GB = $2.5
Total Monthly Cost		$252.96	$240.58

What if I need more than 10 GB of storage?

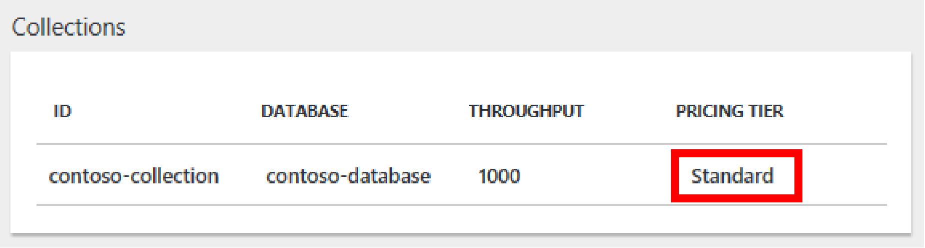
Whether you have a collection with an S1, S2, or S3 performance level, or have a single partition collection, all of which have 10 GB of storage available, you can use the Cosmos DB Data Migration tool to migrate your data to a partitioned collection with virtually unlimited storage. For information about the benefits of a partitioned collection, see Partitioning and scaling in Azure Cosmos DB. For information about how to migrate your S1, S2, S3, or single partition collection to a partitioned collection, see Migrating from single-partition to partitioned collections.

Can I change between the S1, S2, and S3 performance levels before August 1, 2017?

Only existing accounts with S1, S2, and S3 performance will be able to change and alter performance level tiers through the portal or programmatically. By August 1, 2017, the S1, S2, and S3 performance levels will no longer be available. If you change from S1, S3, or S3 to a single partition collection, you cannot return to the S1, S2, or S3 performance levels.

How will I know when my collection has migrated?

The migration will occur on July 31, 2017. If you have a collection that uses the S1, S2 or S3 performance levels, the Cosmos DB team will contact you by email before the migration takes place. Once the migration is complete, on August 1, 2017, the Azure portal will show that your collection uses Standard pricing.



ID	DATABASE	THROUGHPUT	PRICING TIER
contoso-collection	contoso-database	1000	Standard

How do I migrate from the S1, S2, S3 performance levels to single partition collections on my own?

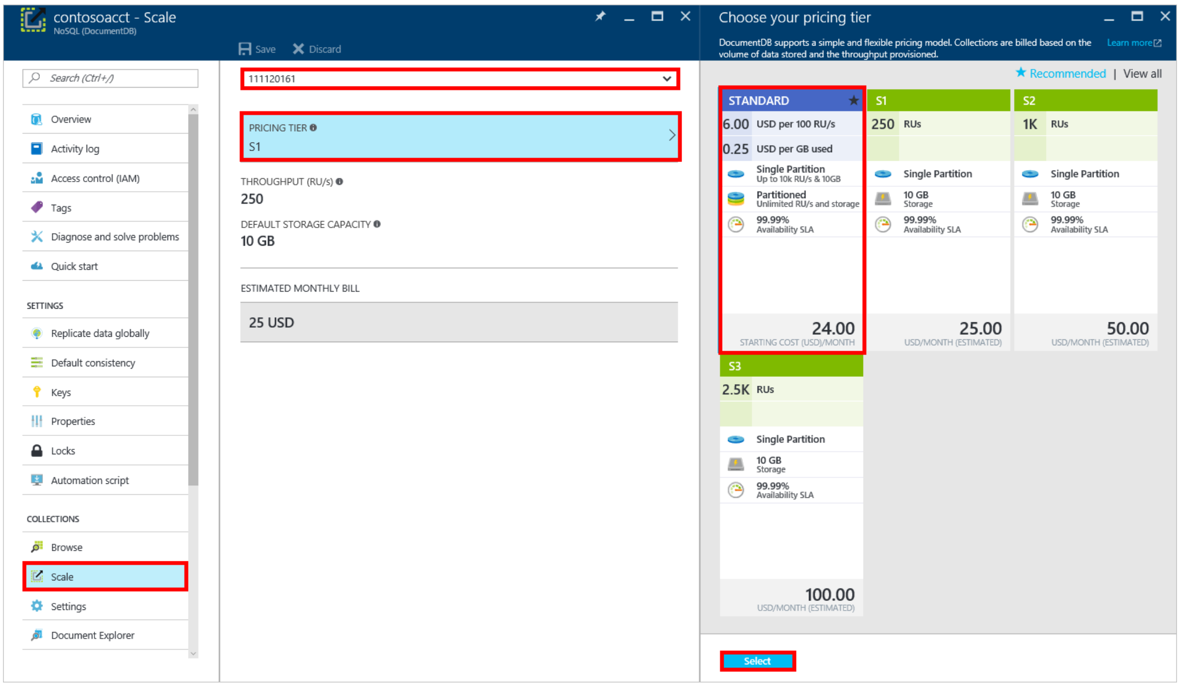
You can migrate from the S1, S2, and S3 performance levels to single partition collections using the Azure portal or programmatically. You can do this on your own before August 1 to benefit from the flexible throughput options available with single partition collections, or we will migrate your collections for you on July 31, 2017.

To migrate to single partition collections using the Azure portal

1. In the **Azure portal**, click **Azure Cosmos DB**, then select the Cosmos DB account to modify.

   If **Azure Cosmos DB** is not on the Jumpbar, click >, scroll to **Databases**, select **Azure Cosmos DB**, and then select the DocumentDB account.

2. On the resource menu, under **Containers**, 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.



111120161

PRICING TIER
S1

THROUGHPUT (RU/s)
250

DEFAULT STORAGE CAPACITY
10 GB

ESTIMATED MONTHLY BILL
25 USD

DocumentDB supports a simple and flexible pricing model. Collections are billed based on the volume of data stored and the throughput provisioned.

STANDARD
6.00 USD per 100 RU/s
0.25 USD per GB used
Single Partition Up to 10k RU/s & 10GB
Partitioned Unlimited RU/s and storage
99.99% Availability SLA
24.00 STARTING COST (USD)/MONTH

S1
250 RUs
Single Partition
10 GB Storage
99.99% Availability SLA
25.00 USD/MONTH (ESTIMATED)

S2
1K RUs
Single Partition
10 GB Storage
99.99% Availability SLA
50.00 USD/MONTH (ESTIMATED)

S3
2.5K RUs
Single Partition
10 GB Storage
99.99% Availability SLA
100.00 USD/MONTH (ESTIMATED)

3. 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.

> IMPORTANT
>
> Once you save your changes and move to the Standard pricing tier, you cannot roll back to the S1, S2, or S3 performance levels.

4. 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 migrate a single partition collection to a partitioned collection, see Migrating from single-partition to partitioned collections.

> NOTE
>
> Changing from S1, S2, or S3 to Standard may take up to 2 minutes.

To migrate to single partition collections 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, visit our getting started tutorial.

Here is a code snippet for changing the collection throughput to 5,000 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);
```

Visit MSDN to view additional examples and learn more about our offer methods:

- **ReadOfferAsync**
- **ReadOffersFeedAsync**
- **ReplaceOfferAsync**
- **CreateOfferQuery**

How am I impacted if I'm an EA customer?

EA customers will be price protected until the end of their current contract.

Next steps

To learn more about pricing and managing data with Azure Cosmos DB, explore these resources:

1. Partitioning data in Cosmos DB. Understand the difference between single partition container and partitioned containers, as well as tips on implementing a partitioning strategy to scale seamlessly.
2. Cosmos DB pricing. Learn about the cost of provisioning throughput and consuming storage.
3. Request units. Understand the consumption of throughput for different operation types, for example Read, Write, Query.

Connect a MongoDB application to Azure Cosmos DB

6/13/2017 • 2 min to read • Edit Online

Learn how to connect your MongoDB app to an Azure Cosmos DB account using a MongoDB connection string. By connecting your MongoDB app to an Azure Cosmos DB database, you can use an Azure Cosmos DB database as the data store for your MongoDB app.

This tutorial provides two ways to retrieve connection string information:

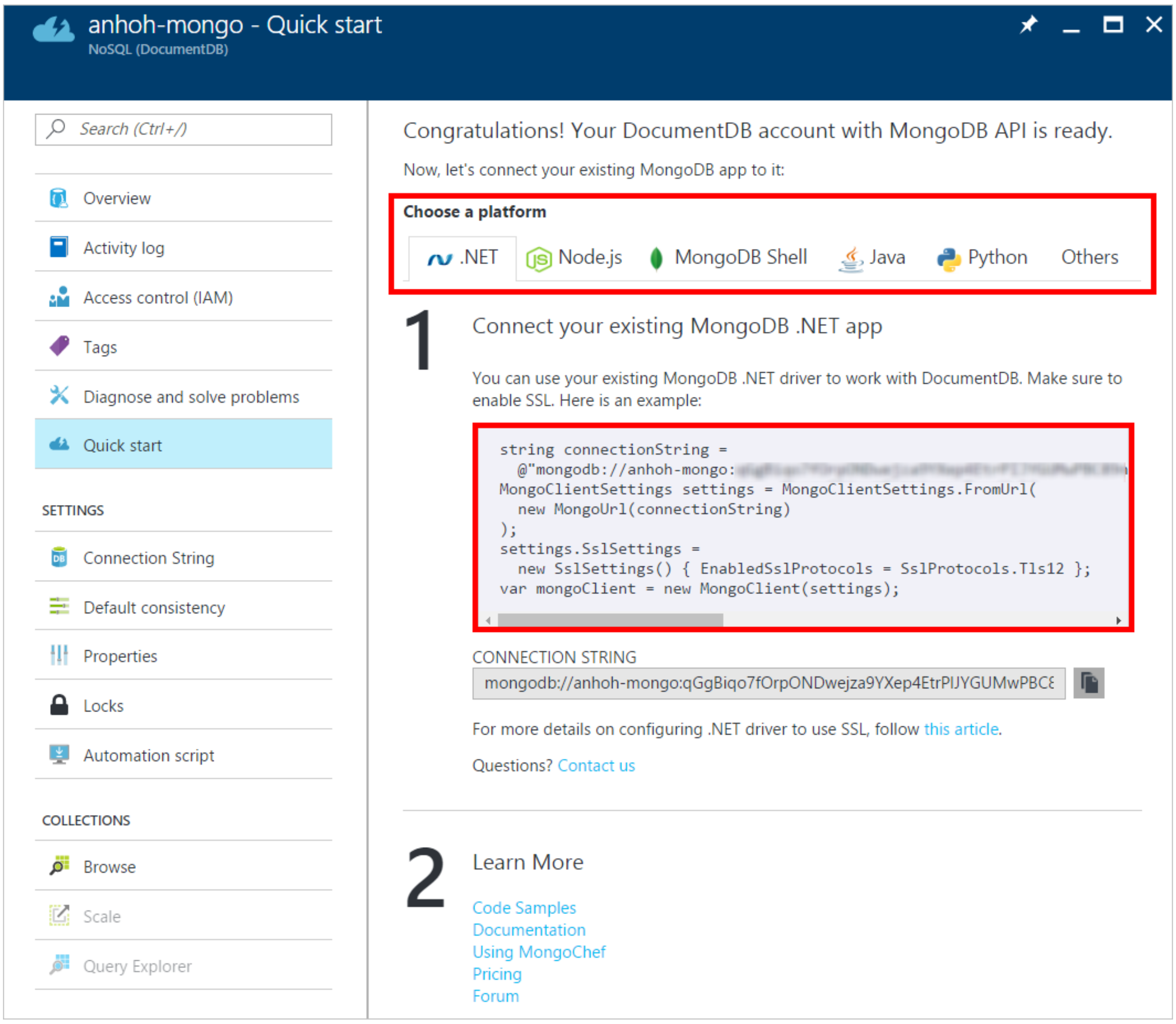
- The Quick start method, for use with .NET, Node.js, MongoDB Shell, Java, and Python drivers.
- The custom connection string method, for use with other drivers.

Prerequisites

- An Azure account. If you don't have an Azure account, create a free Azure account now.
- An Azure Cosmos DB account. For instructions, see Build a MongoDB API web app with .NET and the Azure portal.

Get the MongoDB connection string using the Quick start

1. In an internet browser, sign in to the Azure Portal.
2. In the **Azure Cosmos DB** blade, select the MongoDB API account.
3. In the **Left Navigation** bar of the account blade, click **Quick start**.
4. Choose your platform (*.NET driver*, *Node.js driver*, *MongoDB Shell*, *Java driver*, *Python driver*). If you don't see your driver or tool listed, don't worry, we continuously document more connection code snippets. Please comment below on what you'd like to see and read Get the account's connection string information to learn how to craft your own connection.
5. Copy and paste the code snippet into your MongoDB app, and you are ready to go.



Congratulations! Your DocumentDB account with MongoDB API is ready.

Now, let's connect your existing MongoDB app to it:

1 Connect your existing MongoDB .NET app

You can use your existing MongoDB .NET driver to work with DocumentDB. Make sure to enable SSL. Here is an example:

```
string connectionString =
  @"mongodb://anhoh-mongo:[redacted]
MongoClientSettings settings = MongoClientSettings.FromUrl(
  new MongoUrl(connectionString)
);
settings.SslSettings =
  new SslSettings() { EnabledSslProtocols = SslProtocols.Tls12 };
var mongoClient = new MongoClient(settings);
```

CONNECTION STRING

mongodb://anhoh-mongo:qGgBiqo7fOrpONDwejza9YXep4EtrPIJYGUMwPBC8

For more details on configuring .NET driver to use SSL, follow this article.

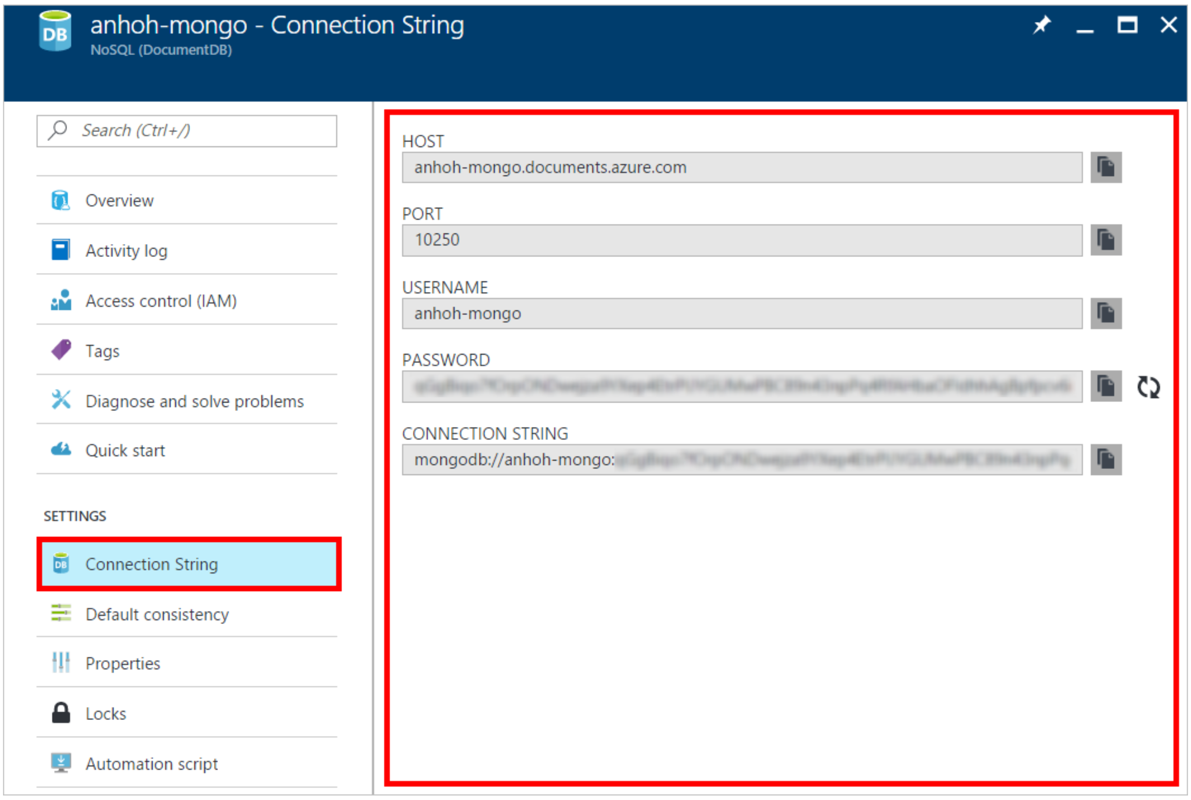
Questions? Contact us

2 Learn More

Code Samples
Documentation
Using MongoChef
Pricing
Forum

Get the MongoDB connection string to customize

1. In an internet browser, sign in to the Azure Portal.
2. In the **Azure Cosmos DB** blade, select the MongoDB API account.
3. In the **Left Navigation** bar of the account blade, click **Connection String**.
4. 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.



HOST
anhoh-mongo.documents.azure.com

PORT
10250

USERNAME
anhoh-mongo

PASSWORD
[redacted]

CONNECTION STRING
mongodb://anhoh-mongo:[redacted]

Connection string requirements

IMPORTANT

Azure Cosmos DB has strict security requirements and standards. Azure Cosmos DB accounts require authentication and secure communication via **SSL**.

It is important to note that Azure Cosmos DB supports the standard MongoDB connection string URI format, with a couple of specific requirements: Azure Cosmos DB 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)
  - Azure Cosmos DB account name
- Password (required)
  - Azure Cosmos DB account password
- Host (required)
  - FQDN of Azure Cosmos DB account
- Port (required)
  - 10255
- Database (optional)
  - The default database used by the connection (if no database is provided, the default database is "test")
- ssl=true (required)

For example, consider the account shown in the Connection String Information above. A valid connection string is:

```
mongodb://contoso123:0Fc3IolnL12312asdfawejunASDF@asdfYXX2t8a97kghVcUzcDv98hawelufhawefafnoQRGwNj2nMPL1Y9qsIr9Srdw==@
anhohmongo.documents.azure.com:10255/mydatabase?ssl=true
```

Next steps

- Learn how to use MongoChef with an Azure Cosmos DB: API for MongoDB account.
- Explore Azure Cosmos DB: API for MongoDB samples.

Use MongoChef with an Azure Cosmos DB: API for MongoDB account

5/30/2017 • 2 min to read • Edit Online

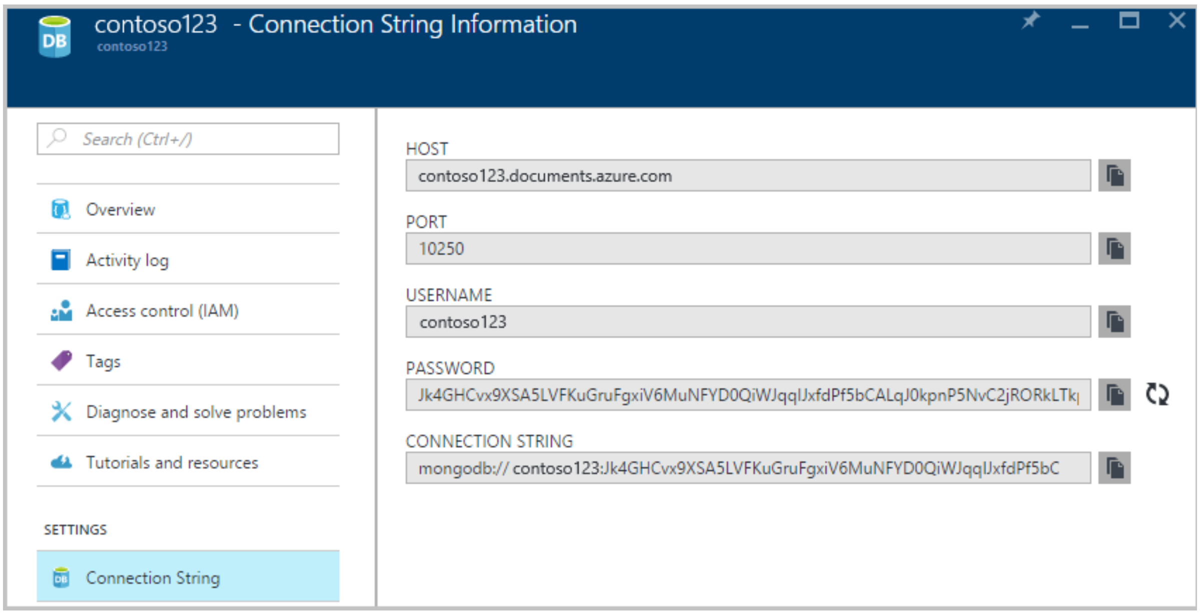
To connect to an Azure Cosmos DB: API for MongoDB account, you must:

- Download and install MongoChef
- Have your Azure Cosmos DB: API for MongoDB account connection string information

Create the connection in MongoChef

To add your Azure Cosmos DB: API for MongoDB account to the MongoChef connection manager, perform the following steps.

1. Retrieve your Azure Cosmos DB: API for MongoDB connection information using the instructions here.



HOST
contoso123.documents.azure.com

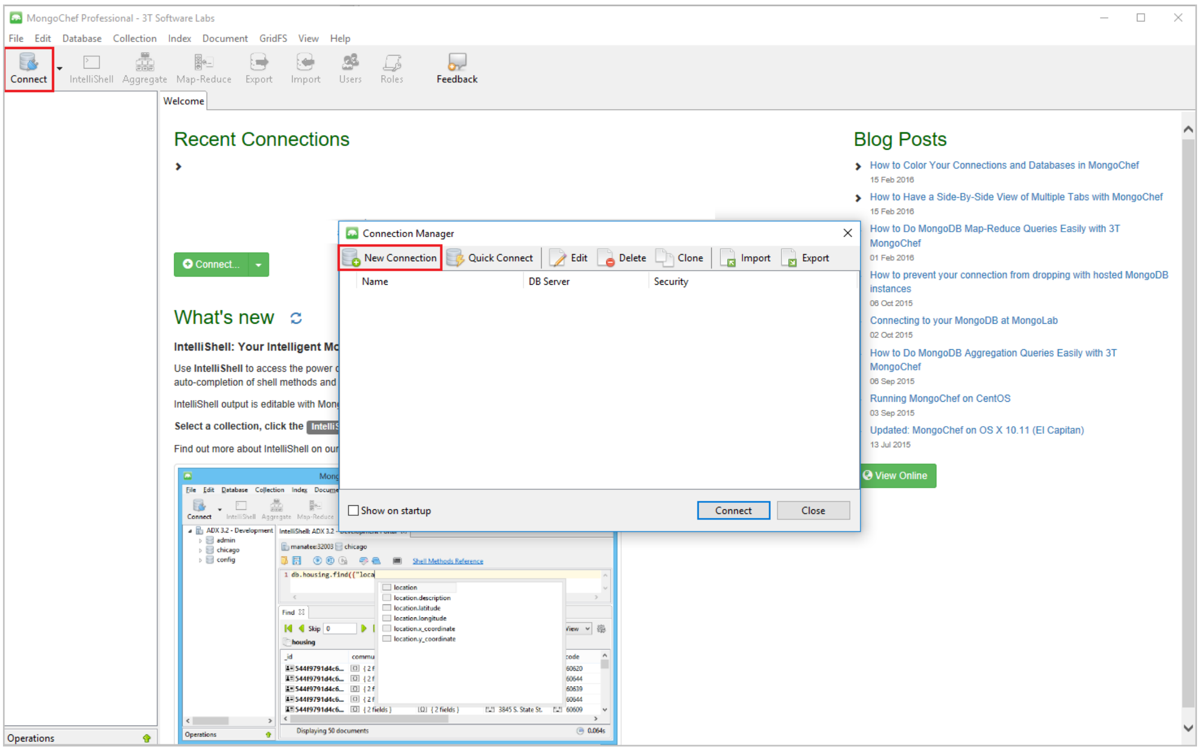
PORT
10250

USERNAME
contoso123

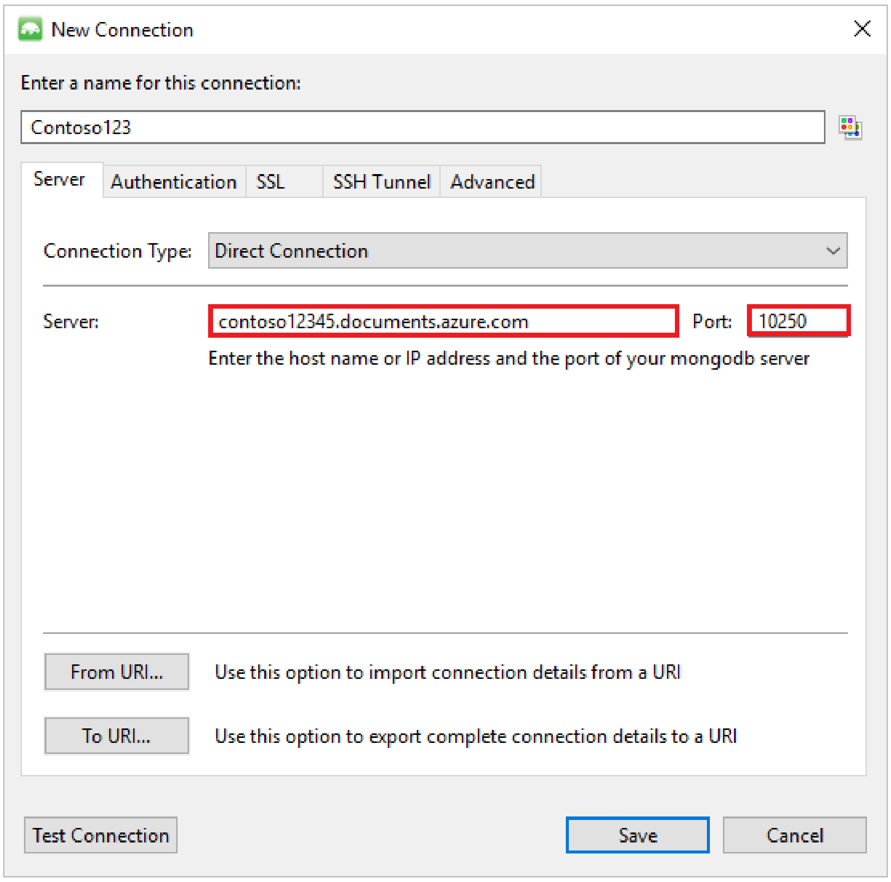
PASSWORD
Jk4GHCvx9XSA5LVFKuGruFgxiV6MuNFYD0QiWJqqIJxfdPf5bCALqJ0kpnP5NvC2jRORkLTkj

CONNECTION STRING
mongodb:// contoso123:Jk4GHCvx9XSA5LVFKuGruFgxiV6MuNFYD0QiWJqqIJxfdPf5bC

2. Click **Connect** to open the Connection Manager, then click **New Connection**


MongoChef Professional - 3T Software Labs
File Edit Database Collection Index Document GridFS View Help
Connect
IntelliShell
Aggregate
Map-Reduce
Export
Import
Users
Roles
Feedback
Welcome
Recent Connections
Connect...
What's new
Blog Posts
How to Color Your Connections and Databases in MongoChef
15 Feb 2016
How to Have a Side-By-Side View of Multiple Tabs with MongoChef
15 Feb 2016
How to Do MongoDB Map-Reduce Queries Easily with 3T MongoChef
01 Feb 2016
How to prevent your connection from dropping with hosted MongoDB instances
06 Oct 2015
Connecting to your MongoDB at MongoLab
02 Oct 2015
How to Do MongoDB Aggregation Queries Easily with 3T MongoChef
06 Sep 2015
Running MongoChef on CentOS
03 Sep 2015
Updated: MongoChef on OS X 10.11 (El Capitan)
13 Jul 2015
View Online
Connection Manager
New Connection
Quick Connect
Edit
Delete
Clone
Import
Export
Name
DB Server
Security
Show on startup
Connect
Close
Operations

3. In the **New Connection** window, on the **Server** tab, enter the HOST (FQDN) of the Azure Cosmos DB: API for MongoDB account and the PORT.



Enter a name for this connection: Contoso123

Connection Type: Direct Connection

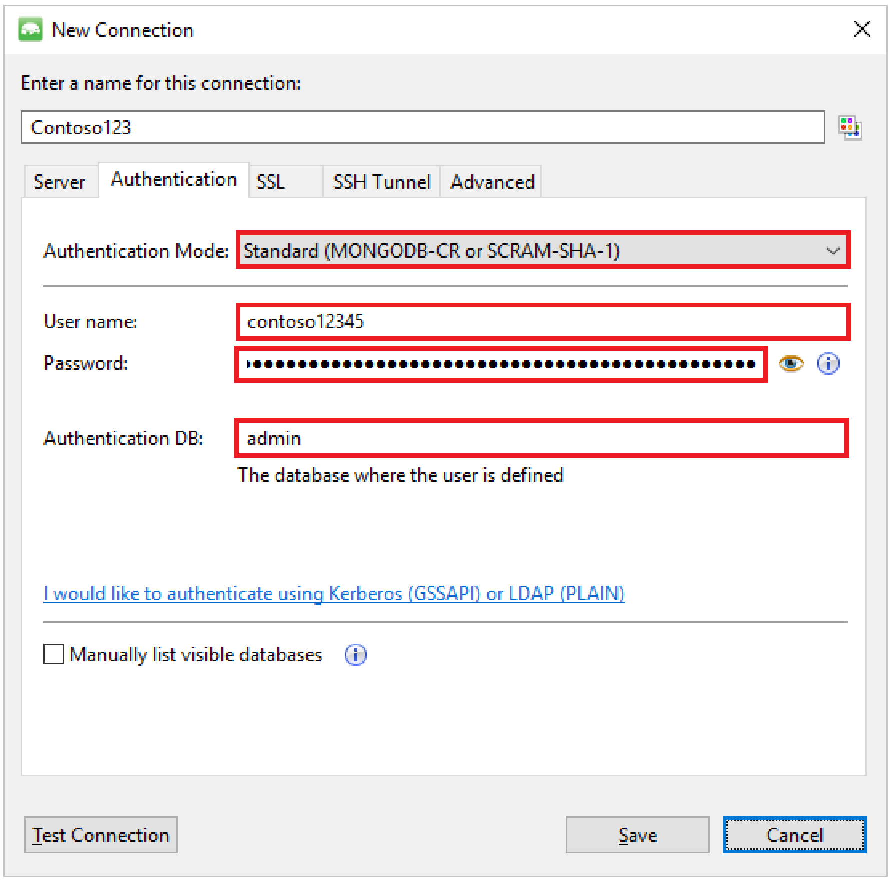
Server: contoso12345.documents.azure.com Port: 10250

Enter the host name or IP address and the port of your mongodb server

Use this option to import connection details from a URI

Use this option to export complete connection details to a URI

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

Authentication Mode: Standard (MONGODB-CR or SCRAM-SHA-1)

User name: contoso12345

Password: ••

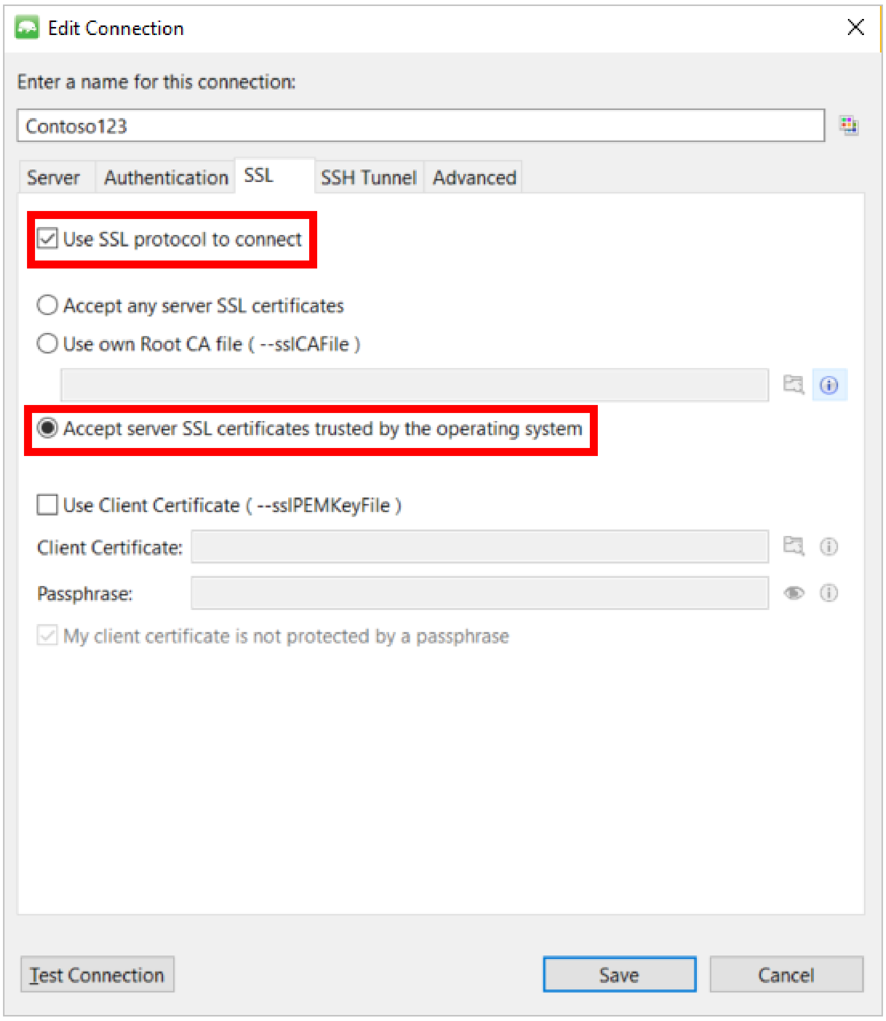
Authentication DB: admin

The database where the user is defined

I would like to authenticate using Kerberos (GSSAPI) or LDAP (PLAIN)

□ Manually list visible databases

5. In the **New Connection** window, on the **SSL** tab, check the **Use SSL protocol to connect** check box and the **Accept server self-signed SSL certificates** radio button.


Edit Connection

Enter a name for this connection:

Contoso123

☑ Use SSL protocol to connect

○ Accept any server SSL certificates

○ Use own Root CA file (--sslCAFile)

● Accept server SSL certificates trusted by the operating system

□ Use Client Certificate (--sslPEMKeyFile)

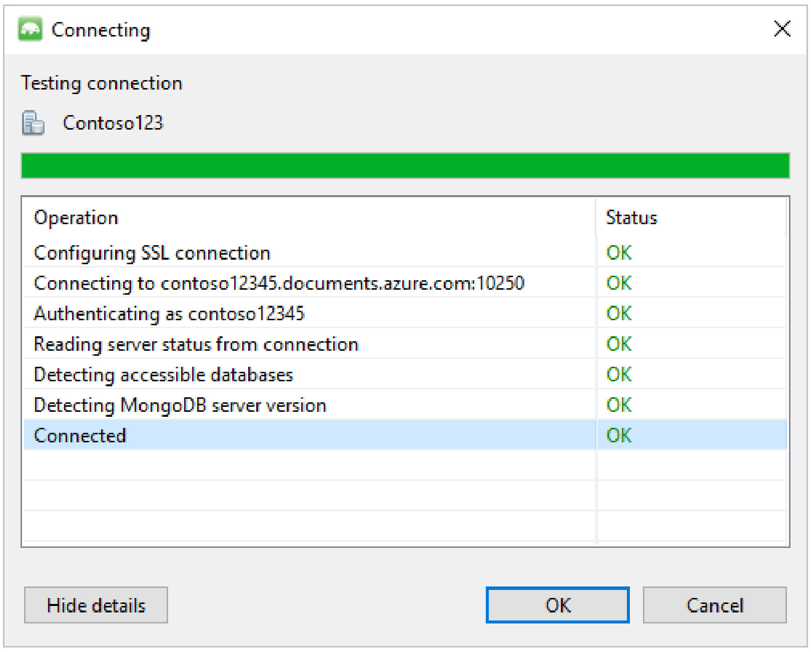
Client Certificate:

Passphrase:

☑ My client certificate is not protected by a passphrase

6. Click the **Test Connection** button to validate the connection information, click **OK** to return to the New

Connection window, and then click **Save**.



Connecting

Testing connection

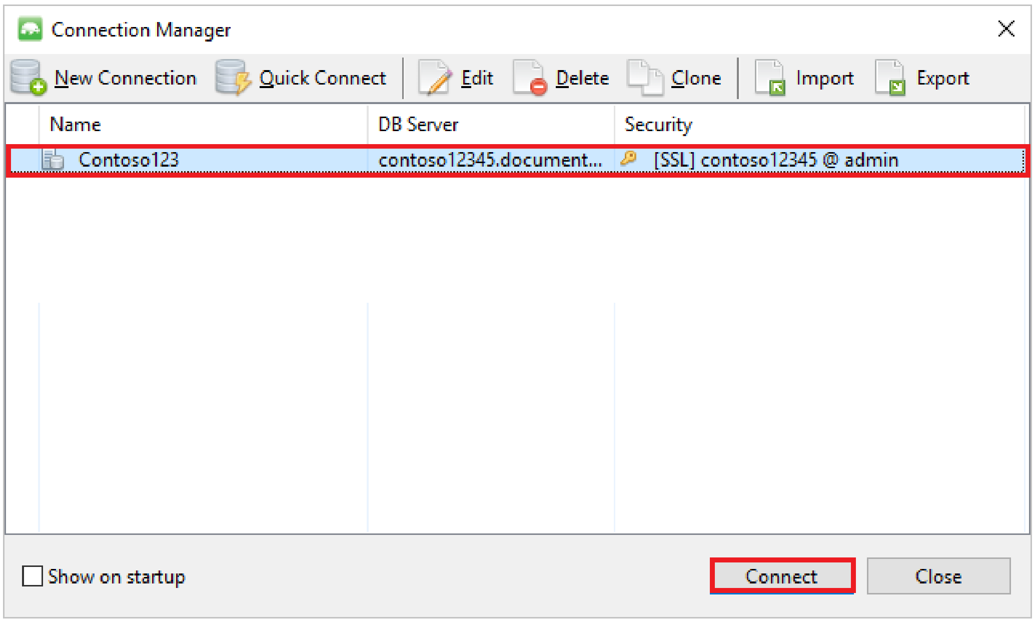
Contoso123

Operation	Status
Configuring SSL connection	OK
Connecting to contoso12345.documents.azure.com:10250	OK
Authenticating as contoso12345	OK
Reading server status from connection	OK
Detecting accessible databases	OK
Detecting MongoDB server version	OK
Connected	OK

Use MongoChef to create a database, collection, and documents

To create a database, collection, and documents using MongoChef, perform the following steps.

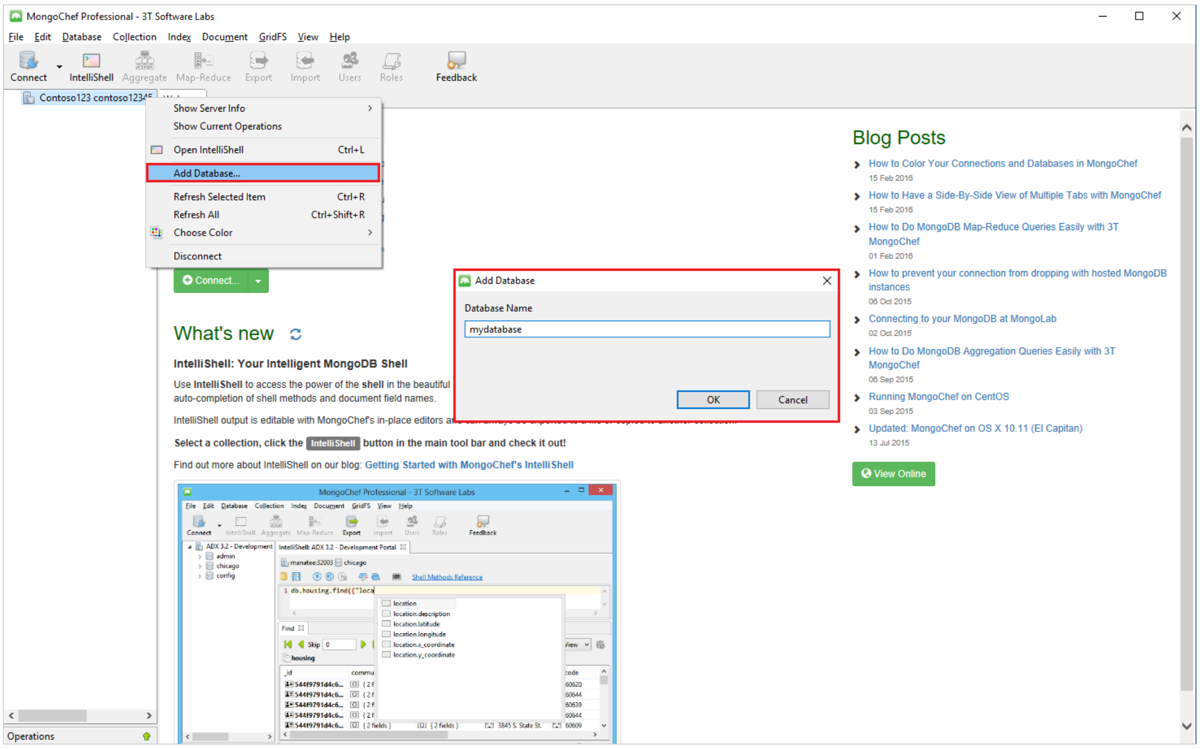
1. In **Connection Manager**, highlight the connection and click **Connect**.



Connection Manager

Name	DB Server	Security
Contoso123	contoso12345.document...	[SSL] contoso12345 @ admin

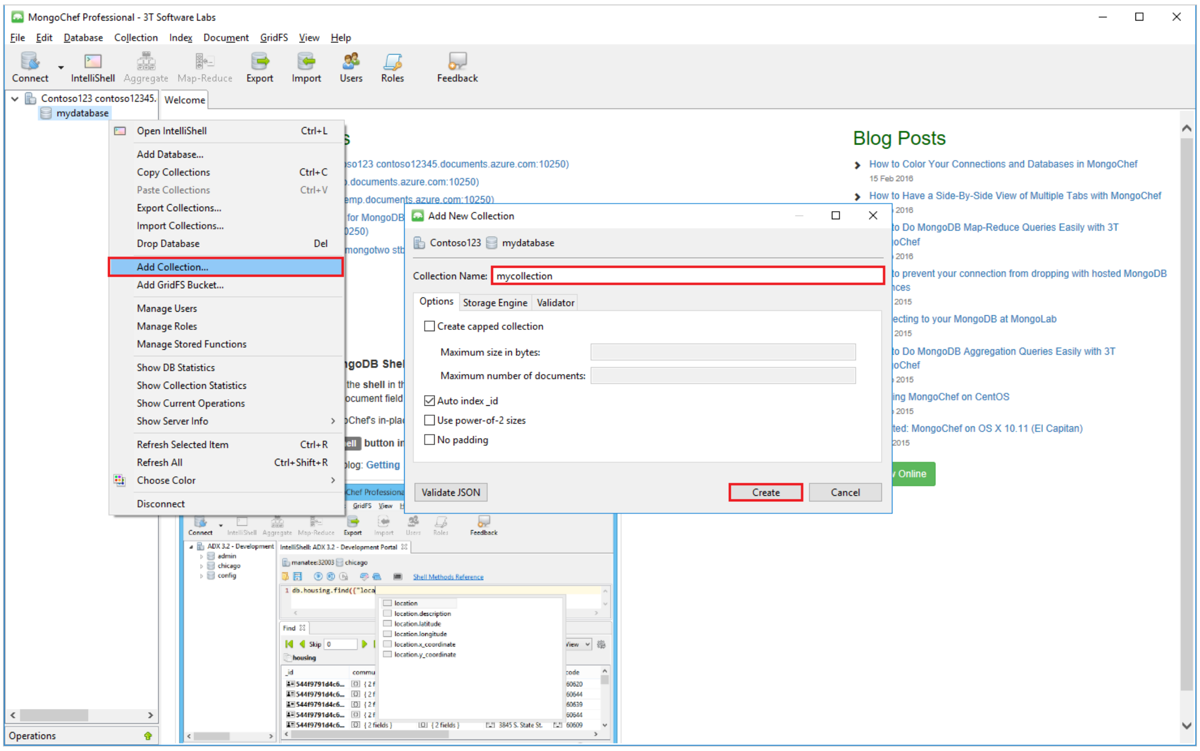
2. Right click the host and choose **Add Database**. Provide a database name and click **OK**.



Add Database

Database Name: mydatabase

3. Right click the database and choose **Add Collection**. Provide a collection name and click **Create**.



Add New Collection

Contoso123 mydatabase

Collection Name: mycollection

□ Create capped collection

Maximum size in bytes:

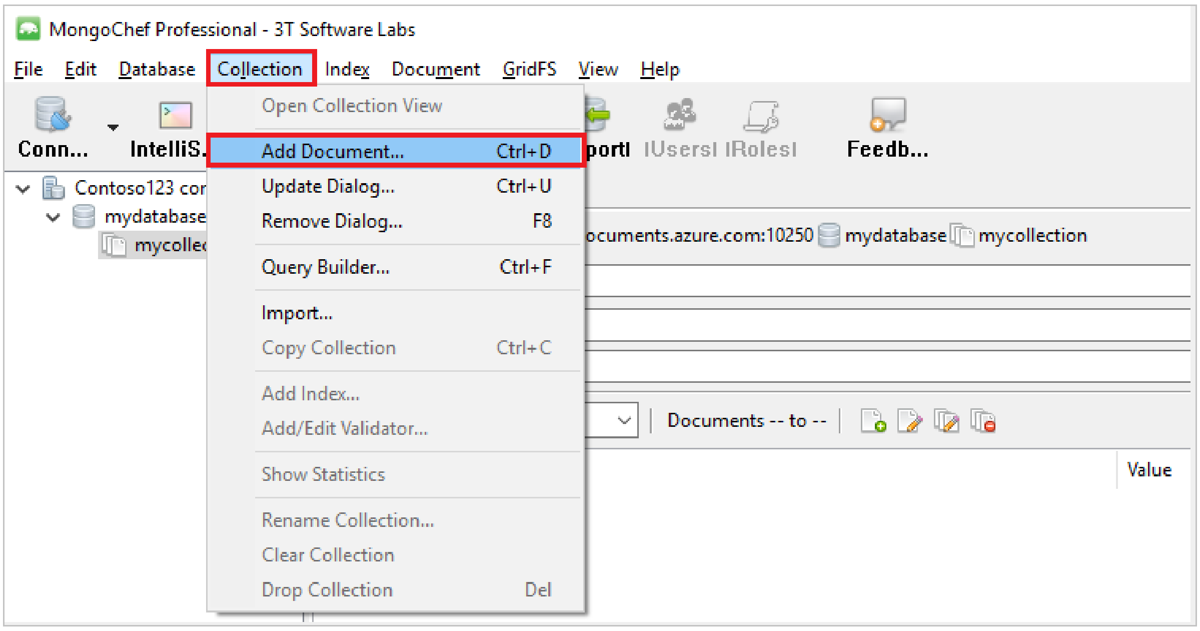
Maximum number of documents:

☑ Auto index _id

□ Use power-of-2 sizes

□ No padding

4. Click the **Collection** menu item, then click **Add Document**.



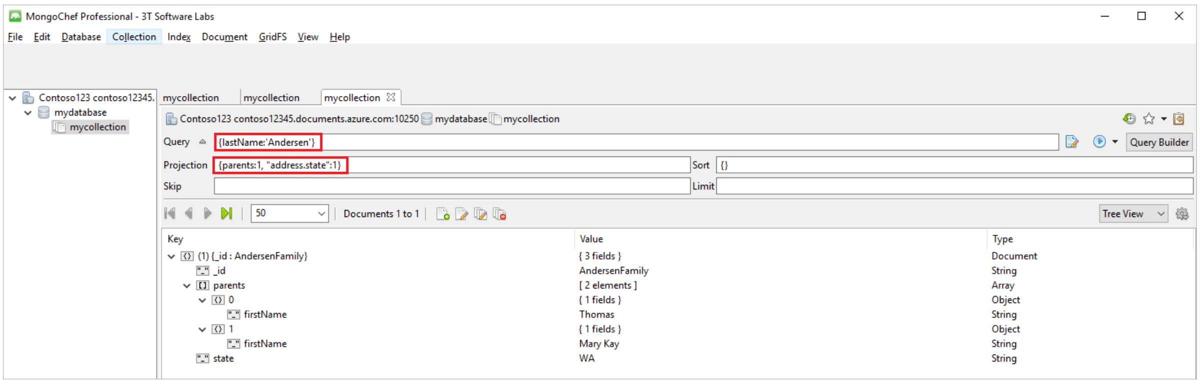
5. In the Add Document dialog, paste the following and then click **Add Document**.

```
{
"_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
}
```

7. Execute a sample query. For example, search for families with the last name 'Andersen' and return the parents and state fields.


MongoChef Professional - 3T Software Labs
File Edit Database Collection Index Document GridFS View Help
Contoso123 contoso12345.
mydatabase
mycollection
mycollection
mycollection
mycollection
Contoso123 contoso12345.documents.azure.com:10250 mydatabase mycollection
Query Builder
Documents 1 to 1
Tree View

Query {lastName:'Andersen'}

Projection {parents:1, "address.state":1}

Sort {}

Skip

Limit

50

Key	Value	Type
(1) {_id : AndersenFamily}	{ 3 fields }	Document
_id	AndersenFamily	String
parents	[2 elements]	Array
0	{ 1 fields }	Object
firstName	Thomas	String
1	{ 1 fields }	Object
firstName	Mary Kay	String
state	WA	String

Next steps

- Explore Azure Cosmos DB: API for MongoDB samples.

Use Robomongo with an Azure Cosmos DB: API for MongoDB account

5/30/2017 • 1 min to read • Edit Online

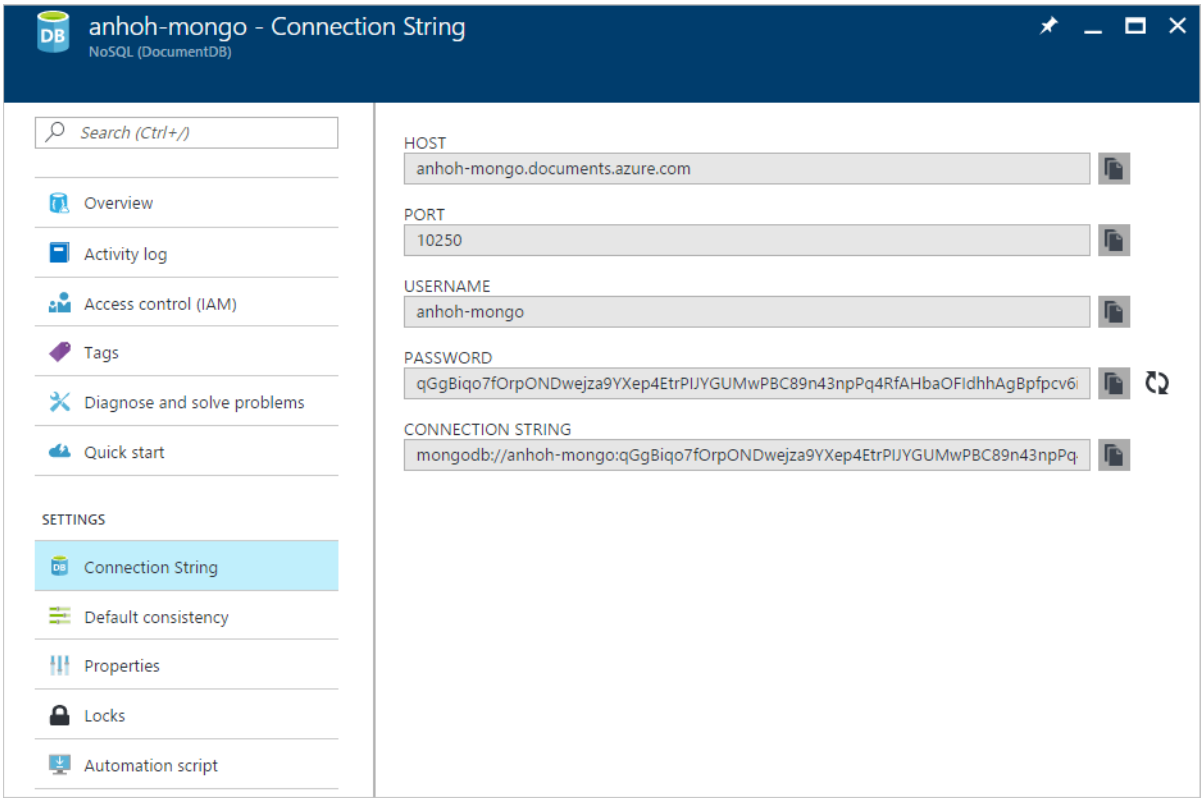
To connect to an Azure Cosmos DB: API for MongoDB account using Robomongo, you must:

- Download and install Robomongo
- Have your Azure Cosmos DB: API for MongoDB account connection string information

Connect using Robomongo

To add your Azure Cosmos DB: API for MongoDB account to the Robomongo MongoDB Connections, perform the following steps.

1. Retrieve your Azure Cosmos DB: API for MongoDB account connection information using the instructions here.



HOST
anhoh-mongo.documents.azure.com

PORT
10250

USERNAME
anhoh-mongo

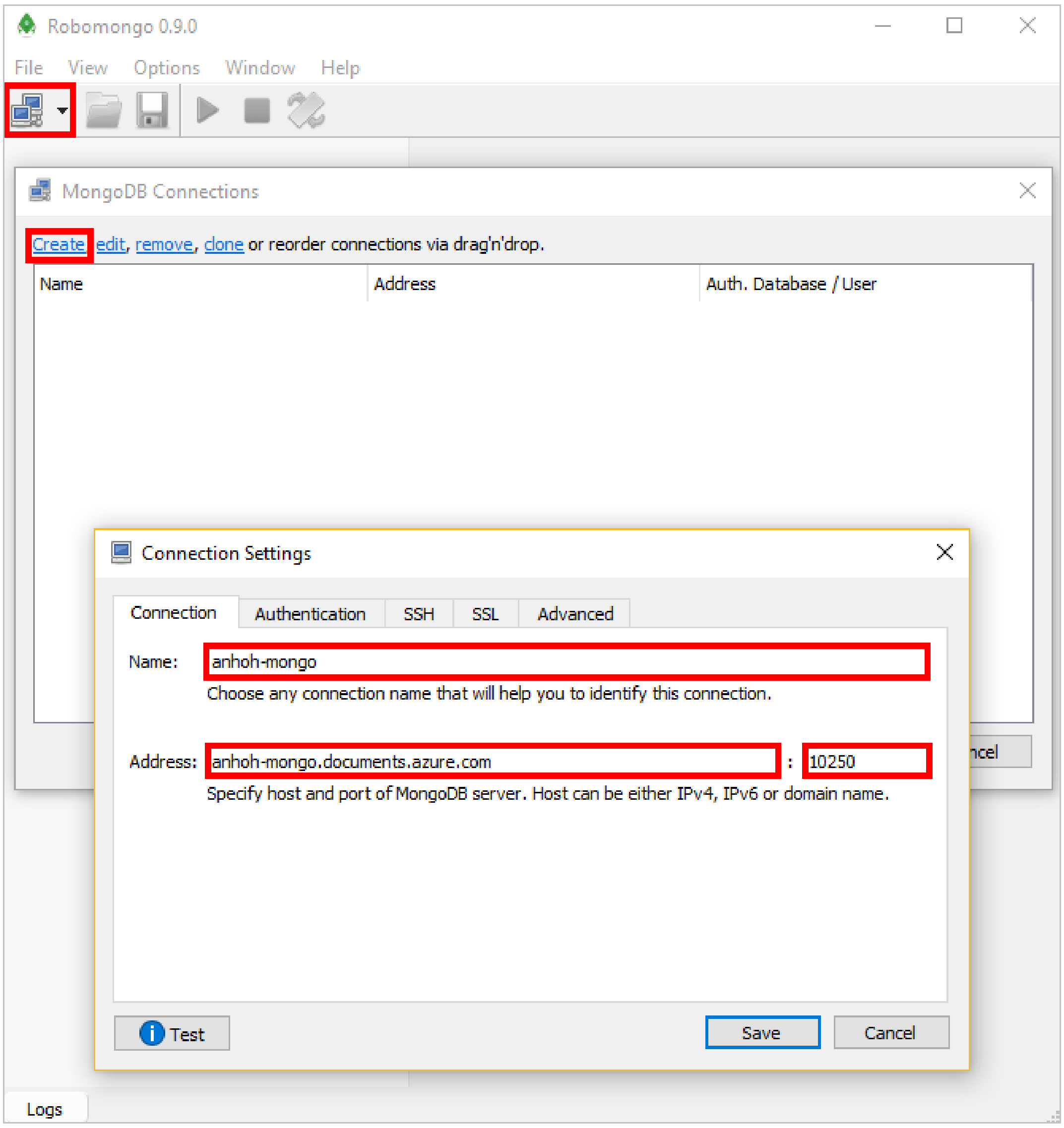
PASSWORD
qGgBiqo7fOrpONDwejza9YXep4EtrPIJYGUMwPBC89n43npPq4RfAHbaOFIdhhAgBpfpcv6i

CONNECTION STRING
mongodb://anhoh-mongo:qGgBiqo7fOrpONDwejza9YXep4EtrPIJYGUMwPBC89n43npPq

2. Run *Robomongo.exe*

3. Click the connection button under **File** to manage your connections. Then, click **Create** in the **MongoDB Connections** window, which will open up the **Connection Settings** window.

4. In the **Connection Settings** window, choose a name. Then, find the **Host** and **Port** from your connection information in Step 1 and enter them into **Address** and **Port**, respectively.



Robomongo 0.9.0
File View Options Window Help
MongoDB Connections
Create, edit, remove, clone or reorder connections via drag'n'drop.
Name Address Auth. Database / User
Connection Settings
Connection Authentication SSH SSL Advanced
Test Save Cancel
Logs

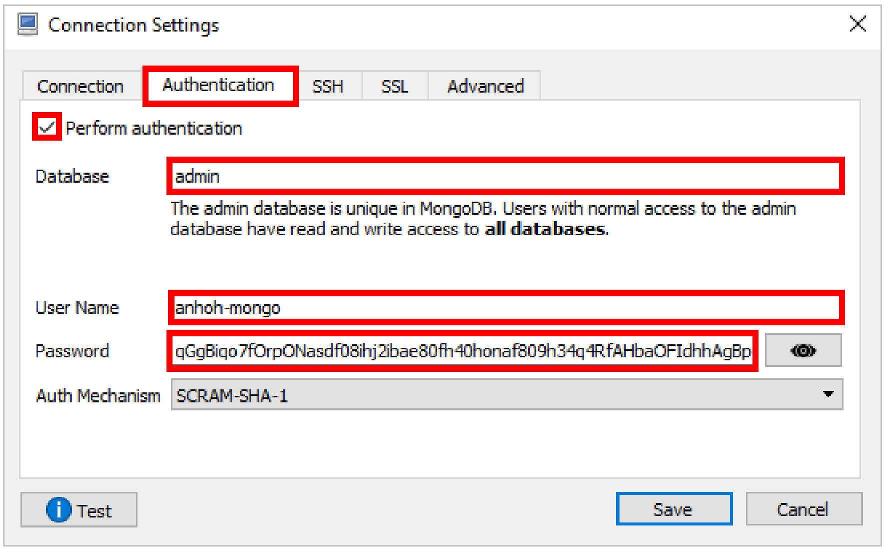
Name: anhoh-mongo

Choose any connection name that will help you to identify this connection.

Address: anhoh-mongo.documents.azure.com : 10250

Specify host and port of MongoDB server. Host can be either IPv4, IPv6 or domain name.

5. On the **Authentication** tab, click **Perform authentication**. Then, enter your Database (default is *Admin*), **User Name** and **Password**. Both **User Name** and **Password** can be found in your connection information in Step 1.



☑ Perform authentication

Database: admin

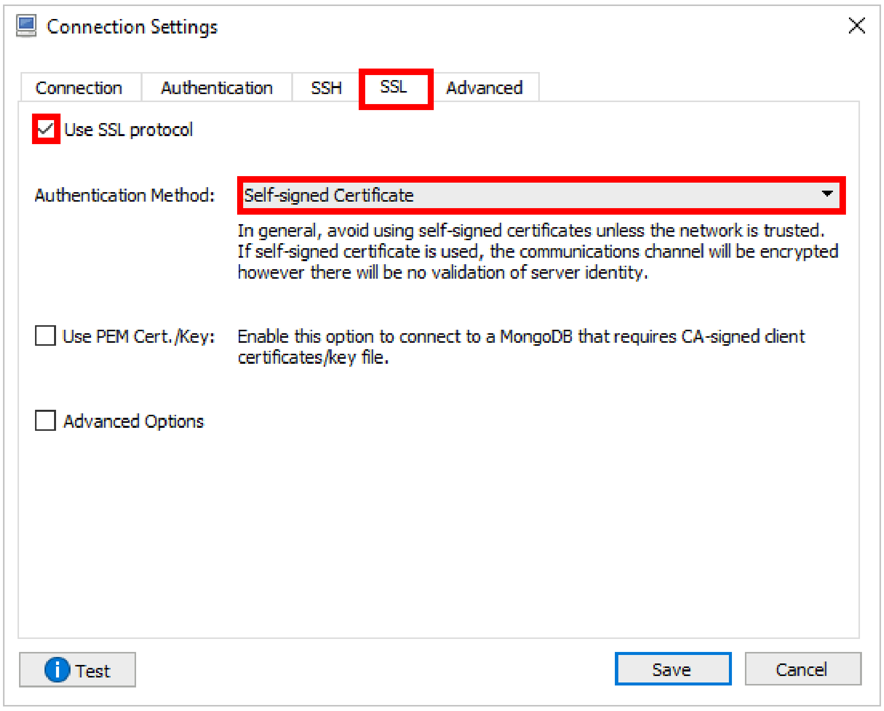
The admin database is unique in MongoDB. Users with normal access to the admin database have read and write access to **all databases**.

User Name: anhoh-mongo

Password: qGgBiqo7fOrpONasdf08ihj2ibae80fh40honaf809h34q4RfAHbaOFIdhhAgBp

Auth Mechanism: SCRAM-SHA-1

6. On the **SSL** tab, check **Use SSL protocol**, then change the **Authentication Method** to **Self-signed Certificate**.



☑ Use SSL protocol

Authentication Method: Self-signed Certificate

In general, avoid using self-signed certificates unless the network is trusted. If self-signed certificate is used, the communications channel will be encrypted however there will be no validation of server identity.

□ Use PEM Cert./Key: Enable this option to connect to a MongoDB that requires CA-signed client certificates/key file.

□ Advanced Options

7. Finally, click **Test** to verify that you are able to connect, then **Save**.

Next steps

- Explore Azure Cosmos DB: API for MongoDB samples.

Build an Azure Cosmos DB: API for MongoDB app using Node.js

5/30/2017 • 1 min to read • Edit Online

This example shows you how to build an Azure Cosmos DB: API for MongoDB console app using Node.js.

To use this example, you must:

- Create an Azure Cosmos DB: API for MongoDB account.
- Retrieve your MongoDB connection string information.

Create the app

1. Create a *app.js* file and copy & paste the code below.

```
var MongoClient = require('mongodb').MongoClient;
var assert = require('assert');
var ObjectId = require('mongodb').ObjectID;
var url = 'mongodb://<endpoint>:<password>@<endpoint>.documents.azure.com:10250/?ssl=true';

var insertDocument = function(db, callback) {
db.collection('families').insertOne( {
    "id": "AndersenFamily",
    "lastName": "Andersen",
    "parents": [
      { "firstName": "Thomas" },
      { "firstName": "Mary Kay" }
    ],
    "children": [
      { "firstName": "John", "gender": "male", "grade": 7 }
    ],
    "pets": [
      { "givenName": "Fluffy" }
    ],
    "address": { "country": "USA", "state": "WA", "city": "Seattle" }
  }, function(err, result) {
  assert.equal(err, null);
  console.log("Inserted a document into the families collection.");
  callback();
});
};

var findFamilies = function(db, callback) {
var cursor =db.collection('families').find( );
cursor.each(function(err, doc) {
  assert.equal(err, null);
  if (doc != null) {
    console.dir(doc);
  } else {
    callback();
  }
});
};

var updateFamilies = function(db, callback) {
db.collection('families').updateOne(
  { "lastName" : "Andersen" },
  {
    $set: { "pets": [
            { "givenName": "Fluffy" },
            { "givenName": "Rocky"}
        ] },
        $currentDate: { "lastModified": true }
    }, function(err, results) {
    console.log(results);
    callback();
});
};

var removeFamilies = function(db, callback) {
db.collection('families').deleteMany(
    { "lastName": "Andersen" },
    function(err, results) {
        console.log(results);
        callback();
    }
);
};

MongoClient.connect(url, function(err, db) {
assert.equal(null, err);
insertDocument(db, function() {
    findFamilies(db, function() {
    updateFamilies(db, function() {
        removeFamilies(db, function() {
            db.close();
        });
    });
    });
});
});
```

2. Modify the following variables in the *app.js* file per your account settings (Learn how to find your connection string):

```
var url = 'mongodb://<endpoint>:<password>@<endpoint>.documents.azure.com:10250/?ssl=true';
```

3. Open your favorite terminal, run **npm install mongodb --save**, then run your app with **node app.js**

Next steps

- Learn how to use MongoChef with your Azure Cosmos DB: API for MongoDB account.

Azure Cosmos DB Gremlin graph support

6/12/2017 • 7 min to read • Edit Online

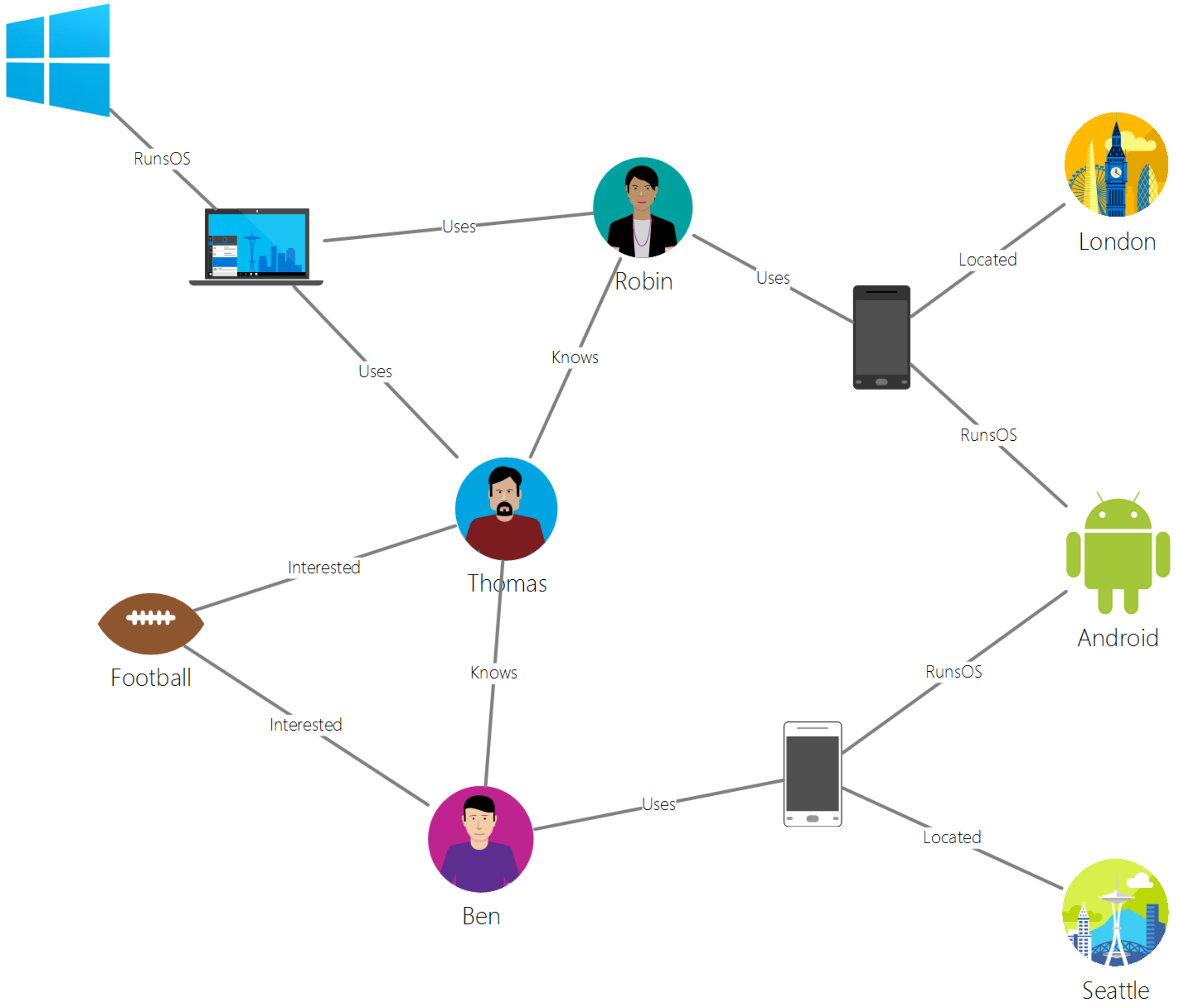
Azure Cosmos DB supports Apache Tinkerpop's graph traversal language, Gremlin which is a Graph API for creating graph entities, and performing graph query operations. You can use the Gremlin language to create graph entities (vertices and edges), modify properties within those entities, perform queries and traversals, and delete entities.

Azure Cosmos DB brings enterprise-ready features to graph databases. This includes global distribution, independent scaling of storage and throughput, predictable single-digit millisecond latencies, automatic indexing, and 99.99% SLAs. Because Azure Cosmos DB supports TinkerPop/Gremlin, you can easily migrate applications written using another graph database without having to make code changes. Additionally, by virtue of Gremlin support, Azure Cosmos DB seamlessly integrates with TinkerPop-enabled analytics frameworks like Apache Spark GraphX.

In this article, we provide a quick walkthrough of Gremlin, and enumerate the Gremlin features and steps that are supported in the preview of Graph API support.

Gremlin by example

Let's use a sample graph to understand how queries can be expressed in Gremlin. The following figure shows a business application that manages data about users, interests, and devices in the form of a graph.



This graph has the following vertex types (called "label" in Gremlin):

- People: the graph has three people, Robin, Thomas, and Ben
- Interests: their interests, in this example, the game of Football
- Devices: the devices that people use
- Operating Systems: the operating systems that the devices run on

We represent the relationships between these entities via the following edge types/labels:

- Knows: For example, "Thomas knows Robin"
- Interested: To represent the interests of the people in our graph, for example, "Ben is interested in Football"
- RunsOS: Laptop runs the Windows OS
- Uses: To represent which device a person uses. For example, Robin uses a Motorola phone with serial number 77

Let's run some operations against this graph using the Gremlin Console. You can also perform these operations using Gremlin drivers in the platform of your choice (Java, Node.js, Python, or .NET). Before we look at what's supported in Azure Cosmos DB, let's look at a few examples to get familiar with the syntax.

First let's look at CRUD. The following Gremlin statement inserts the "Thomas" vertex into the graph:

```
:> g.addV('person').property('id', 'thomas.1').property('firstName', 'Thomas').property('lastName', 'Andersen').property('age', 44)
```

Next, the following Gremlin statement inserts a "knows" edge between Thomas and Robin.

```
:> g.V('thomas.1').addE('knows').to(g.V('robin.1'))
```

The following query returns the "person" vertices in descending order of their first names:

```
:> g.V().hasLabel('person').order().by('firstName', decr)
```

Where graphs shine is when you need to answer questions like "What operating systems do friends of Thomas use?". You can run this simple Gremlin traversal to get that information from the graph:

```
:> g.V('thomas.1').out('knows').out('uses').out('runsos').group().by('name').by(count())
```

Now let's look at what Azure Cosmos DB provides for Gremlin developers.

Gremlin features

TinkerPop is a standard that covers a wide range of graph technologies. Therefore, it has standard terminology to describe what features are provided by a graph provider. Azure Cosmos DB provides a persistent, high concurrency, writeable graph database that can be partitioned across multiple servers or clusters.

The following table lists the TinkerPop features that are implemented by Azure Cosmos DB:

CATEGORY	AZURE COSMOS DB IMPLEMENTATION	NOTES
Graph features	Provides Persistence and ConcurrentAccess in preview. Designed to support Transactions	Computer methods can be implemented via the Spark connector.
Variable features	Supports Boolean, Integer, Byte, Double, Float, Integer, Long, String	Supports primitive types, is compatible with complex types via data model
Vertex features	Supports RemoveVertices, MetaProperties, AddVertices, MultiProperties, StringIds, UserSuppliedIds, AddProperty, RemoveProperty	Supports creating, modifying, and deleting vertices
Vertex property features	StringIds, UserSuppliedIds, AddProperty, RemoveProperty, BooleanValues, ByteValues, DoubleValues, FloatValues, IntegerValues, LongValues, StringValues	Supports creating, modifying, and deleting vertex properties
Edge features	AddEges, RemoveEdges, StringIds, UserSuppliedIds, AddProperty, RemoveProperty	Supports creating, modifying, and deleting edges
Edge property features	Properties, BooleanValues, ByteValues, DoubleValues, FloatValues, IntegerValues, LongValues, StringValues	Supports creating, modifying, and deleting edge properties

Gremlin wire format: GraphSON

Azure Cosmos DB uses the GraphSON format when returning results from Gremlin operations. GraphSON is the Gremlin standard format for representing vertices, edges, and properties (single and multi-valued properties) using JSON.

For example, the following snippet shows a GraphSON representation of a vertex *returned to the client* from Azure Cosmos DB.

```
{
 "id": "a7111ba7-0ea1-43c9-b6b2-efc5e3aea4c0",
 "label": "person",
 "type": "vertex",
 "outE": {
  "knows": [
   {
    "id": "3ee53a60-c561-4c5e-9a9f-9c7924bc9aef",
    "inV": "04779300-1c8e-489d-9493-50fd1325a658"
   },
   {
    "id": "21984248-ee9e-43a8-a7f6-30642bc14609",
    "inV": "a8e3e741-2ef7-4c01-b7c8-199f8e43e3bc"
   }
  ]
 },
 "properties": {
  "firstName": [
   {
    "value": "Thomas"
   }
  ],
  "lastName": [
   {
    "value": "Andersen"
   }
  ],
  "age": [
   {
    "value": 45
   }
  ]
 }
}
```

The properties used by GraphSON for vertices are the following:

PROPERTY	DESCRIPTION
id	The ID for the vertex. Must be unique (in combination with the value of _partition if applicable)
label	The label of the vertex. This is optional, and used to describe the entity type.
type	Used to distinguish vertices from non-graph documents
properties	Bag of user-defined properties associated with the vertex. Each property can have multiple values.
_partition (configurable)	The partition key of the vertex. Can be used to scale out graphs to multiple servers

PROPERTY	DESCRIPTION
outE	This contains a list of out edges from a vertex. Storing the adjacency information with vertex allows for fast execution of traversals. Edges are grouped based on their labels.

And the edge contains the following information to help with navigation to other parts of the graph.

PROPERTY	DESCRIPTION
id	The ID for the edge. Must be unique (in combination with the value of _partition if applicable)
label	The label of the edge. This property is optional, and used to describe the relationship type.
inV	Bag of user-defined properties associated with the edge. Each property can have multiple values.
properties	Bag of user-defined properties associated with the edge. Each property can have multiple values.

Each property can store multiple values within an array.

PROPERTY	DESCRIPTION
value	The value of the property

Gremlin partitioning

In Azure Cosmos DB, graphs are stored within containers that can scale independently in terms of storage and throughput (expressed in normalized requests per second). Each container must define an optional, but recommended partition key property that determines a logical partition boundary for related data. Every vertex/edge must have an `id` property that is unique for entities within that partition key value. The details are covered in Partitioning in Azure Cosmos DB.

Gremlin operations work seamlessly across graph data that span multiple partitions in Azure Cosmos DB. However, it is recommended to choose a partition key for your graphs that is commonly used as a filter in queries, has many distinct values, and similar frequency of access these values.

Gremlin steps

Now let's look at the Gremlin steps supported by Azure Cosmos DB. For a complete reference on Gremlin, see TinkerPop reference.

STEP	DESCRIPTION	TINKERPOP 3.2 DOCUMENTATION	NOTES
`addE`	Adds an edge between two vertices	addE step	
`addV`	Adds a vertex to the graph	addV step	

STEP	DESCRIPTION	TINKERPOP 3.2 DOCUMENTATION	NOTES
`and`	Ensurest that all the traversals return a value	and step	
`as`	A step modulator to assign a variable to the output of a step	as step	
`by`	A step modulator used with `group` and `order`	by step	
`coalesce`	Returns the first traversal that returns a result	coalesce step	
`constant`	Returns a constant value. Used with `coalesce`	constant step	
`count`	Returns the count from the traversal	count step	
`dedup`	Returns the values with the duplicates removed	dedup step	
`drop`	Drops the values (vertex/edge)	drop step	
`fold`	Acts as a barrier that computes the aggregate of results	fold step	
`group`	Groups the values based on the labels specified	group step	
`has`	Used to filter properties, vertices, and edges. Supports `hasLabel`, `hasId`, `hasNot`, and `has` variants.	has step	
`inject`	Inject values into a stream	inject step	
`is`	Used to perform a filter using a boolean expression	is step	
`limit`	Used to limit number of items in the traversal	limit step	
`local`	Local wraps a section of a traversal, similar to a subquery	local step	
`not`	Used to produce the negation of a filter	not step	

STEP	DESCRIPTION	TINKERPOP 3.2 DOCUMENTATION	NOTES
`optional`	Returns the result of the specified traversal if it yields a result else it returns the calling element	optional step	
`or`	Ensures at least one of the traversals returns a value	or step	
`order`	Returns results in the specified sort order	order step	
`path`	Returns the full path of the traversal	path step	
`project`	Projects the properties as a Map	project step	
`properties`	Returns the properties for the specified labels	properties step	
`range`	Filters to the specified range of values	range step	
`repeat`	Repeats the step for the specified number of times. Used for looping	repeat step	
`sample`	Used to sample results from the traversal	sample step	
`select`	Used to project results from the traversal	select step	
`store`	Used for non-blocking aggregates from the traversal	store step	
`tree`	Aggregate paths from a vertex into a tree	tree step	
`unfold`	Unroll an iterator as a step	unfold step	
`union`	Merge results from multiple traversals	union step	
`V`	Includes the steps necessary for traversals between vertices and edges `V`, `E`, `out`, `in`, `both`, `outE`, `inE`, `bothE`, `outV`, `inV`, `bothV`, and `otherV` for	vertex steps	

STEP	DESCRIPTION	TINKERPOP 3.2 DOCUMENTATION	NOTES
`where`	Used to filter results from the traversal. Supports `eq` , `neq` , `lt` , `lte` , `gt` , `gte` , and `between` operators	where step	

Azure Cosmos DB's write-optimized engine supports automatic indexing of all properties within vertices and edges by default. Therefore, queries with filters, range queries, sorting, or aggregates on any property are processed from the index, and served efficiently. For more information on how indexing works in Azure Cosmos DB, see our paper on schema-agnostic indexing.

Next Steps

- Get started building a graph application using our SDKs
- Learn more about Azure Cosmos DB's graph support

Working with the change feed support in Azure Cosmos DB

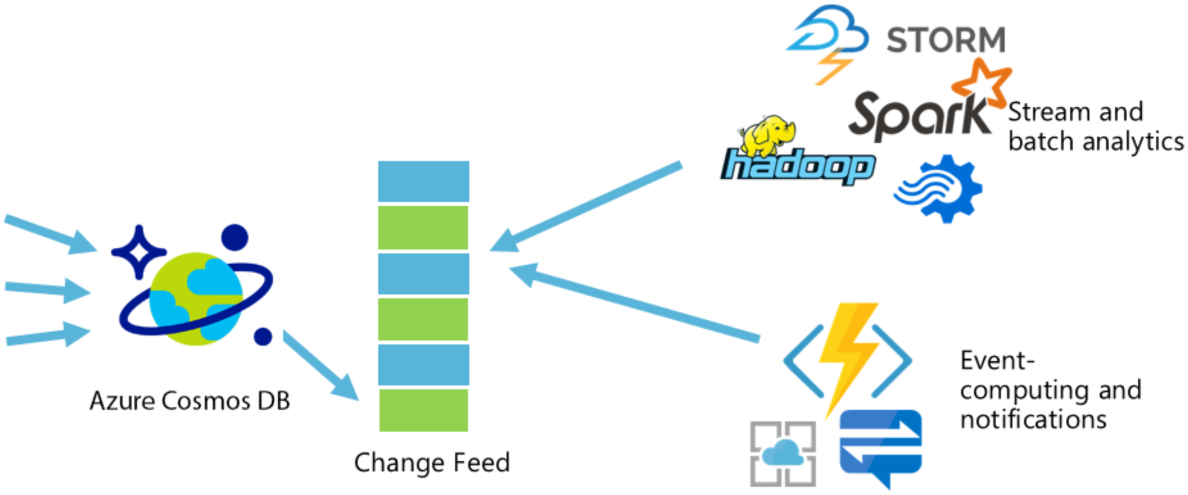
5/30/2017 • 13 min to read • Edit Online

Azure Cosmos DB is a fast and flexible globally replicated database service that is used for storing high-volume transactional and operational data with predictable single-digit millisecond latency for reads and writes. This makes it well-suited for IoT, gaming, retail, and operational logging applications. A common design pattern in these applications is to track changes made to Azure Cosmos DB data, and update materialized views, perform real-time analytics, archive data to cold storage, and trigger notifications on certain events based on these changes. The **change feed support** in Azure Cosmos DB enables you to build efficient and scalable solutions for each of these patterns.

With change feed support, Azure Cosmos DB provides a sorted list of documents within an Azure Cosmos DB collection in the order in which they were modified. This feed can be used to listen for modifications to data within the collection and perform actions such as:

- Trigger a call to an API when a document is inserted or modified
- Perform real-time (stream) processing on updates
- Synchronize data with a cache, search engine, or data warehouse

Changes in Azure Cosmos DB are persisted and can be processed asynchronously, and distributed across one or more consumers for parallel processing. Let's look at the APIs for change feed and how you can use them to build scalable real-time applications. This article shows how to work with spatial data with the Azure Cosmos DB DocumentDB API.



Use cases and scenarios

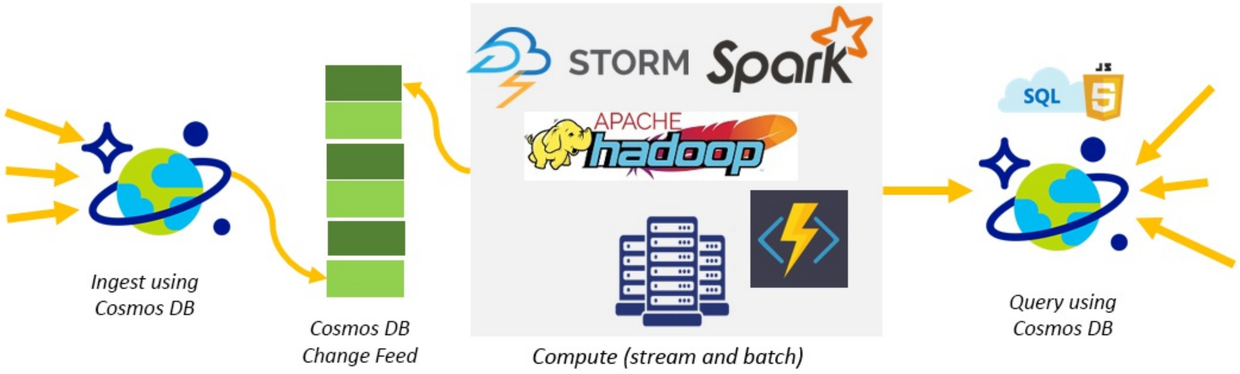
Change feed allows for efficient processing of large datasets with a high volume of writes, and offers an alternative to querying entire datasets to identify what has changed. For example, you can perform the following tasks efficiently:

- Update a cache, search index, or a data warehouse with data stored in Azure Cosmos DB.
- Implement application-level data tiering and archival, that is, store "hot data" in Azure Cosmos DB, and age out

"cold data" to Azure Blob Storage or Azure Data Lake Store.

- Implement batch analytics on data using Apache Hadoop.
- Implement lambda pipelines on Azure with Azure Cosmos DB. Azure Cosmos DB provides a scalable database solution that can handle both ingestion and query, and implement lambda architectures with low TCO.
- Perform zero down-time migrations to another Azure Cosmos DB account with a different partitioning scheme.

Lambda Pipelines with Azure Cosmos DB for ingestion and query:



You can use Azure Cosmos DB to receive and store event data from devices, sensors, infrastructure, and applications, and process these events in real-time with Azure Stream Analytics, Apache Storm, or Apache Spark.

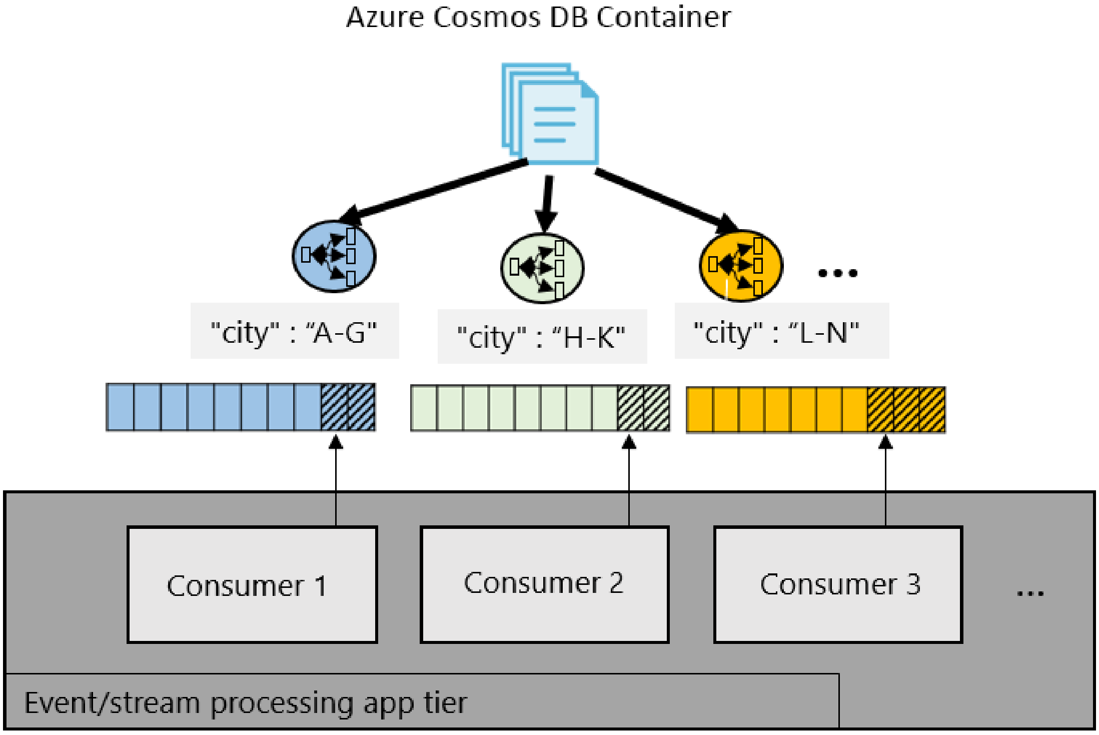
Within web and mobile apps, you can track events such as changes to your customer's profile, preferences, or location to trigger certain actions like sending push notifications to their devices using Azure Functions or App Services. If you're using Azure Cosmos DB to build a game, you can, for example, use change feed to implement real-time leaderboards based on scores from completed games.

How change feed works in Azure Cosmos DB

Azure Cosmos DB provides the ability to incrementally read updates made to an Azure Cosmos DB collection. This change feed has the following properties:

- Changes are persistent in Azure Cosmos DB and can be processed asynchronously.
- Changes to documents within a collection are available immediately in the change feed.
- Each change to a document appears only once in the change feed. Only the most recent change for a given document is included in the change log. Intermediate changes may not be available.
- The change feed is sorted by order of modification within each partition key value. There is no guaranteed order across partition-key values.
- Changes can be synchronized from any point-in-time, that is, there is no fixed data retention period for which changes are available.
- Changes are available in chunks of partition key ranges. This capability allows changes from large collections to be processed in parallel by multiple consumers/servers.
- Applications can request for multiple change feeds simultaneously on the same collection.

Azure Cosmos DB's change feed is enabled by default for all accounts, and does not incur any additional costs on your account. You can use your provisioned throughput in your write region or any read region to read from the change feed, just like any other operation from Azure Cosmos DB. The change feed includes inserts and update operations made to documents within the collection. You can capture deletes by setting a "soft-delete" flag within your documents in place of deletes. Alternatively, you can set a finite expiration period for your documents via the TTL capability, for example, 24 hours and use the value of that property to capture deletes. With this solution, you have to process changes within a shorter time interval than the TTL expiration period. The change feed is available for each partition key range within the document collection, and thus can be distributed across one or more consumers for parallel processing.



In the following section, we describe how to access the change feed using the Azure Cosmos DB REST API and SDKs. For .NET applications, we recommend using the for processing events from the change feed.

Working with the REST API and SDK

Azure Cosmos DB provides elastic containers of storage and throughput called **collections**. Data within collections is logically grouped using partition keys for scalability and performance. Azure Cosmos DB provides various APIs for accessing this data, including lookup by ID (Read/Get), query, and read-feeds (scans). The change feed can be obtained by populating two new request headers to the DocumentDB `ReadDocumentFeed` API, and can be processed in parallel across ranges of partition keys.

ReadDocumentFeed API

Let's take a brief look at how ReadDocumentFeed works. Azure Cosmos DB supports reading a feed of documents within a collection via the `ReadDocumentFeed` API. For example, the following request returns a page of documents inside the `serverlogs` collection.

```
GET https://mydocumentdb.documents.azure.com/dbs/smalldb/colls/serverlogs HTTP/1.1
x-ms-date: Tue, 22 Nov 2016 17:05:14 GMT
authorization: type%3dmaster%26ver%3d1.0%26sig%3dgo7JEogZDn6ritWhwc5hX%2fNTV4wwM1u9V2Is1H4%2bDRg%3d
Cache-Control: no-cache
x-ms-consistency-level: Strong
User-Agent: Microsoft.Azure.Documents.Client/1.10.27.5
x-ms-version: 2016-07-11
Accept: application/json
Host: mydocumentdb.documents.azure.com
```

Results can be limited by using the `x-ms-max-item-count` header, and reads can be resumed by resubmitting the request with a `x-ms-continuation` header returned in the previous response. When performed from a single client, `ReadDocumentFeed` iterates through results across partitions serially.

Serial Read Document Feed

You can also retrieve the feed of documents using one of the supported Azure Cosmos DB SDKs. For example, the following snippet shows how to perform ReadDocumentFeed in .NET.

```
FeedResponse<dynamic> feedResponse = null;
do
{
    feedResponse = await client.ReadDocumentFeedAsync(collection, new FeedOptions { MaxItemCount = -1 });
}
while (feedResponse.ResponseContinuation != null);
```

Distributed execution of ReadDocumentFeed

For collections that contain terabytes of data or more, or ingest a large volume of updates, serial execution of read feed from a single client machine might not be practical. In order to support these big data scenarios, Azure Cosmos DB provides APIs to distribute `ReadDocumentFeed` calls transparently across multiple client readers/consumers.

Distributed Read Document Feed

To provide scalable processing of incremental changes, Azure Cosmos DB supports a scale-out model for the change feed API based on ranges of partition keys.

- You can obtain a list of partition key ranges for a collection performing a `ReadPartitionKeyRanges` call.
- For each partition key range, you can perform a `ReadDocumentFeed` to read documents with partition keys within that range.

Retrieving partition key ranges for a collection

You can retrieve the Partition Key Ranges by requesting the `pkranges` resource within a collection. For example the following request retrieves the list of partition key ranges for the `serverlogs` collection:

```
GET https://querydemo.documents.azure.com/dbs/bigdb/colls/serverlogs/pkranges HTTP/1.1
x-ms-date: Tue, 15 Nov 2016 07:26:51 GMT
authorization: type%3dmaster%26ver%3d1.0%26sig%3dEConYmRgDExu6q%2bZ8GjfUGOH0AcOx%2behkancw3LsGQ8%3d
x-ms-consistency-level: Session
x-ms-version: 2016-07-11
Accept: application/json
Host: querydemo.documents.azure.com
```

This request returns the following response containing metadata about the partition key ranges:

```
HTTP/1.1 200 Ok
Content-Type: application/json
x-ms-item-count: 25
x-ms-schemaversion: 1.1
Date: Tue, 15 Nov 2016 07:26:51 GMT

{
  "_rid":"qYcAAPEvJBQ=",
  "PartitionKeyRanges":[
    {
      "_rid":"qYcAAPEvJBQCAAAAAAAAUA==",
      "id":"0",
      "_etag":"\"00002800-0000-0000-0000-580ac4ea0000\"",
      "minInclusive":"",
      "maxExclusive":"05C1CFFFFFFFF8",
      "_self":"dbs\/qYcAAA==\/colls\/qYcAAPEvJBQ=\/pkranges\/qYcAAPEvJBQCAAAAAAAAUA==\/",
      "_ts":1477100776
    },
    ...
  ],
  "_count": 25
}
```

Partition Key Range Properties: Each partition key range includes the metadata properties in the following table:

HEADER NAME	DESCRIPTION
id	The ID for the partition key range. This is a stable and unique ID within each collection. Must be used in the following call to read changes by partition key range.
maxExclusive	The maximum partition key hash value for the partition key range. For internal use.
minInclusive	The minimum partition key hash value for the partition key range. For internal use.

You can do this using one of the supported Azure Cosmos DB SDKs. For example, the following snippet shows how to retrieve partition key ranges in .NET.

```
string pkRangesResponseContinuation = null;
List<PartitionKeyRange> partitionKeyRanges = new List<PartitionKeyRange>();

do
{
    FeedResponse<PartitionKeyRange> pkRangesResponse = await client.ReadPartitionKeyRangeFeedAsync(
        collectionUri,
        new FeedOptions { RequestContinuation = pkRangesResponseContinuation });

    partitionKeyRanges.AddRange(pkRangesResponse);
    pkRangesResponseContinuation = pkRangesResponse.ResponseContinuation;
}
while (pkRangesResponseContinuation != null);
```

Azure Cosmos DB supports retrieval of documents per partition key range by setting the optional `x-ms-documentdb-partitionkeyrangeid` header.

Performing an incremental ReadDocumentFeed

ReadDocumentFeed supports the following scenarios/tasks for incremental processing of changes in Azure Cosmos DB collections:

- Read all changes to documents from the beginning, that is, from collection creation.
- Read all changes to future updates to documents from current time.
- Read all changes to documents from a logical version of the collection (ETag). You can checkpoint your consumers based on the returned ETag from incremental read-feed requests.

The changes include inserts and updates to documents. To capture deletes, you must use a "soft delete" property within your documents, or use the built-in TTL property to signal a pending deletion in the change feed.

The following table lists the request and response headers for ReadDocumentFeed operations.

Request Headers for incremental ReadDocumentFeed:

HEADER NAME	DESCRIPTION
A-IM	Must be set to "Incremental feed", or omitted otherwise
If-None-Match	No header: returns all changes from the beginning (collection creation) "*": returns all new changes to data within the collection <etag>: If set to a collection ETag, returns all changes made since that logical timestamp
x-ms-documentdb-partitionkeyrangeid	The partition key range ID for reading data.

Response Headers for incremental ReadDocumentFeed:

HEADER NAME	DESCRIPTION
etag	The logical sequence number (LSN) of last document returned in the response. incremental ReadDocumentFeed can be resumed by resubmitting this value in If-None-Match.

Here's a sample request to return all incremental changes in collection from the logical version/ETag `28535` and partition key range = `16`:

```
GET https://mydocumentdb.documents.azure.com/dbs/bigdb/colls/bigcoll/docs HTTP/1.1
x-ms-max-item-count: 1
If-None-Match: "28535"
A-IM: Incremental feed
x-ms-documentdb-partitionkeyrangeid: 16
x-ms-date: Tue, 22 Nov 2016 20:43:01 GMT
authorization: type%3dmaster%26ver%3d1.0%26sig%3dzdpL2QQ8TCfiNbW%2fEcT88JHNvWeCgDA8gWeRZ%2btfN5o%3d
x-ms-version: 2016-07-11
Accept: application/json
Host: mydocumentdb.documents.azure.com
```

Changes are ordered by time within each partition key value within the partition key range. There is no guaranteed order across partition-key values. If there are more results than can fit in a single page, you can read the next page of results by resubmitting the request with the `If-None-Match` header with value equal to the `etag` from the previous response. If multiple documents were inserted or updated transactionally within a stored procedure or trigger, they will all be returned within the same response page.

> NOTE
>
> With change feed, you might get more items returned in a page than specified in `x-ms-max-item-count` in the case of multiple documents inserted or updated inside a stored procedures or triggers.

The .NET SDK provides the CreateDocumentChangeFeedQuery and ChangeFeedOptions helper classes to access changes made to a collection. The following snippet shows how to retrieve all changes from the beginning using the .NET SDK from a single client.

```
private async Task<Dictionary<string, string>> GetChanges(
    DocumentClient client,
    string collection,
    Dictionary<string, string> checkpoints)
{
    string pkRangesResponseContinuation = null;
    List<PartitionKeyRange> partitionKeyRanges = new List<PartitionKeyRange>();

    do
    {
        FeedResponse<PartitionKeyRange> pkRangesResponse = await client.ReadPartitionKeyRangeFeedAsync(
            collectionUri,
            new FeedOptions { RequestContinuation = pkRangesResponseContinuation });

        partitionKeyRanges.AddRange(pkRangesResponse);
        pkRangesResponseContinuation = pkRangesResponse.ResponseContinuation;
    }
    while (pkRangesResponseContinuation != null);

    foreach (PartitionKeyRange pkRange in partitionKeyRanges)
    {
        string continuation = null;
        checkpoints.TryGetValue(pkRange.Id, out continuation);

        IDocumentQuery<Document> query = client.CreateDocumentChangeFeedQuery(
            collection,
            new ChangeFeedOptions
            {
                PartitionKeyRangeId = pkRange.Id,
                StartFromBeginning = true,
                RequestContinuation = continuation,
                MaxItemCount = 1
            });

        while (query.HasMoreResults)
        {
            FeedResponse<DeviceReading> readChangesResponse = query.ExecuteNextAsync<DeviceReading>().Result;

            foreach (DeviceReading changedDocument in readChangesResponse)
            {
                Console.WriteLine(changedDocument.Id);
            }

            checkpoints[pkRange.Id] = readChangesResponse.ResponseContinuation;
        }
    }

    return checkpoints;
}
```

And the following snippet shows how to process changes in real-time with Azure Cosmos DB by using the change feed support and the preceding function. The first call returns all the documents in the collection, and the second only returns the two documents created that were created since the last checkpoint.

```
// Returns all documents in the collection.
Dictionary<string, string> checkpoints = await GetChanges(client, collection, new Dictionary<string, string>());

await client.CreateDocumentAsync(collection, new DeviceReading { DeviceId = "xsensr-201", MetricType = "Temperature", Unit = "Celsius",
MetricValue = 1000 });
await client.CreateDocumentAsync(collection, new DeviceReading { DeviceId = "xsensr-212", MetricType = "Pressure", Unit = "psi",
MetricValue = 1000 });

// Returns only the two documents created above.
checkpoints = await GetChanges(client, collection, checkpoints);
```

You can also filter the change feed using client side logic to selectively process events. For example, here's a snippet that uses client side LINQ to process only temperature change events from device sensors.

```
FeedResponse<DeviceReading> readChangesResponse = query.ExecuteNextAsync<DeviceReading>().Result;

foreach (DeviceReading changedDocument in
    readChangesResponse.AsEnumerable().Where(d => d.MetricType == "Temperature" && d.MetricValue > 1000L))
{
    // trigger an action, like call an API
}
```

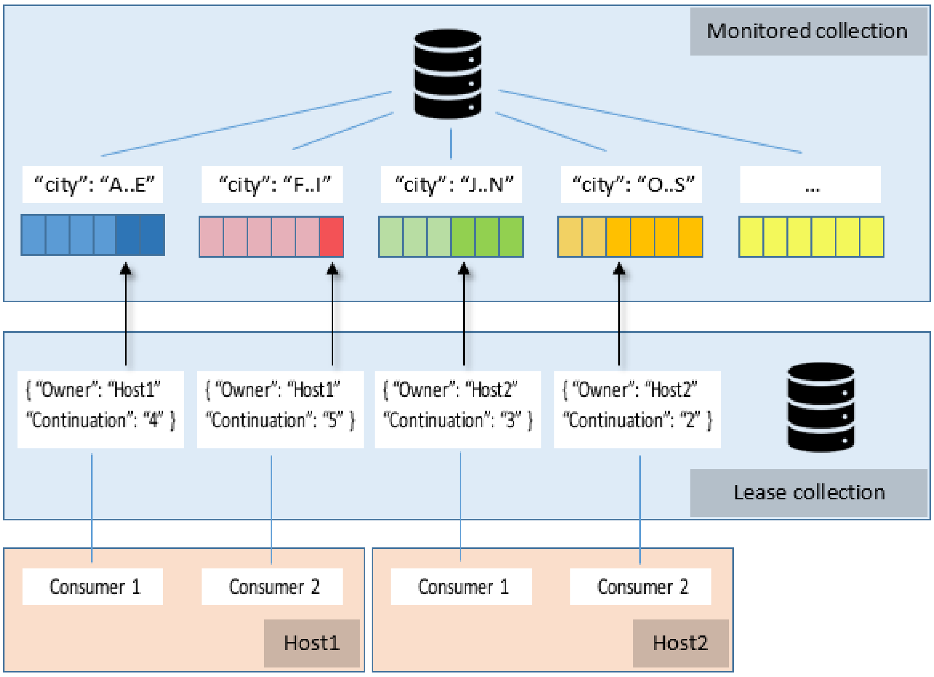
Change feed processor library

The Azure Cosmos DB change feed processor library can be used to distribute event processing from the change feed across multiple consumers. You should use this implementation when building change feed readers on the .NET platform. The `ChangeFeedProcessorHost` class provides a thread-safe, multi-process, safe runtime environment for event processor implementations that also provides checkpointing and partition lease management.

To use the `ChangeFeedProcessorHost` class, you can implement `IChangeFeedObserver`. This interface contains three methods:

- OpenAsync
- CloseAsync
- ProcessEventsAsync

To start event processing, instantiate ChangeFeedProcessorHost, providing the appropriate parameters for your Azure Cosmos DB collection. Then, call `RegisterObserverAsync` to register your `IChangeFeedObserver` implementation with the runtime. At this point, the host will attempt to acquire a lease on every partition key range in the Azure Cosmos DB collection using a "greedy" algorithm. These leases will last for a given timeframe and must then be renewed. As new nodes, worker instances in this case, come online, they place lease reservations and over time the load shifts between nodes as each attempts to acquire more leases.


Monitored collection
"city": "A..E"
"city": "F..I"
"city": "J..N"
"city": "O..S"
...
{ "Owner": "Host1" "Continuation": "4" }
{ "Owner": "Host1" "Continuation": "5" }
{ "Owner": "Host2" "Continuation": "3" }
{ "Owner": "Host2" "Continuation": "2" }
Lease collection
Consumer 1
Consumer 2
Host1
Consumer 1
Consumer 2
Host2

Over time, an equilibrium is established. This dynamic capability enables CPU-based autoscaling to be applied to consumers for both scale-up and scale-down. If changes are available in Azure Cosmos DB at a faster rate than consumers can process, the CPU increase on consumers can be used to cause an auto-scale on worker instance count.

The `ChangeFeedProcessorHost` class also implements an checkpointing mechanism using a separate Azure Cosmos DB leases collection. This mechanism stores the offset on a per-partition basis, so that each consumer can determine what the last checkpoint from the previous consumer was. As partitions transition between nodes via leases, this is the synchronization mechanism that facilitates load shifting.

Here's a code snippet for a simple change feed processor host that prints changes to the console:

```
class DocumentFeedObserver : IChangeFeedObserver
{
    private static int s_totalDocs = 0;
    public Task OpenAsync(ChangeFeedObserverContext context)
    {
        Console.WriteLine("Worker opened, {0}", context.PartitionKeyRangeId);
        return Task.CompletedTask;  // Requires targeting .NET 4.6+.
    }
    public Task CloseAsync(ChangeFeedObserverContext context, ChangeFeedObserverCloseReason reason)
    {
        Console.WriteLine("Worker closed, {0}", context.PartitionKeyRangeId);
        return Task.CompletedTask;
    }
    public Task ProcessEventsAsync(IReadOnlyList<Document> docs, ChangeFeedObserverContext context)
    {
        Console.WriteLine("Change feed: total {0} doc(s)", Interlocked.Add(ref s_totalDocs, docs.Count));
        return Task.CompletedTask;
    }
}
```

The following code snippet shows how to register a new host to listen to changes from an Azure Cosmos DB collection. Here, we configure a separate collection to manage the leases to partitions across multiple consumers:

```
string hostName = Guid.NewGuid().ToString();
DocumentCollectionInfo documentCollectionLocation = new DocumentCollectionInfo
{
    Uri = new Uri("https://YOUR_SERVICE.documents.azure.com:443/"),
    MasterKey = "YOUR_SECRET_KEY==",
    DatabaseName = "db1",
    CollectionName = "documents"
};

DocumentCollectionInfo leaseCollectionLocation = new DocumentCollectionInfo
{
    Uri = new Uri("https://YOUR_SERVICE.documents.azure.com:443/"),
    MasterKey = "YOUR_SECRET_KEY==",
    DatabaseName = "db1",
    CollectionName = "leases"
};

ChangeFeedEventHost host = new ChangeFeedEventHost(hostName, documentCollectionLocation, leaseCollectionLocation);
await host.RegisterObserverAsync<DocumentFeedObserver>();
```

In this article, we provided a walkthrough of Azure Cosmos DB's change feed support, and how to track changes made to Azure Cosmos DB data using the REST API and/or SDKs.

Next steps

- Try the Azure Cosmos DB Change feed code samples on GitHub
- Get started coding with the Azure Cosmos DB SDKs or the REST API

Working with geospatial and GeoJSON location data in Azure Cosmos DB

6/6/2017 • 11 min to read • Edit Online

This article is an introduction to the geospatial functionality in Azure Cosmos DB. After reading this, you will be able to answer the following questions:

- How do I store spatial data in Azure Cosmos DB?
- How can I query geospatial data in Azure Cosmos DB in SQL and LINQ?
- How do I enable or disable spatial indexing in Azure Cosmos DB?

This article shows how to work with spatial data with the DocumentDB API. 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

Azure Cosmos DB 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 Azure Cosmos DB without any specialized tools or libraries. The Azure Cosmos DB 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 Azure Cosmos DB) using its coordinate pair or longitude and latitude. Here's an example JSON for a point.

Points in Azure Cosmos DB

```
{
   "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.
>
> Azure Cosmos DB 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 an Azure Cosmos DB document as shown in this example of a user profile containing location data:

Use Profile with Location stored in Azure Cosmos DB

```
{
   "id":"documentdb-profile",
   "screen_name":"@CosmosDB",
   "city":"Redmond",
   "topics":[ "global", "distributed" ],
   "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 Azure Cosmos DB.

Polygons in GeoJSON

```
{
   "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 Azure Cosmos DB. However Azure Cosmos DB 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. Azure Cosmos DB 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 an Azure Cosmos DB 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 DocumentDB APIs, you can use the `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 Azure Cosmos DB using SQL and LINQ.

Spatial SQL built-in functions

Azure Cosmos DB 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 Azure Cosmos DB.

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 Azure Cosmos DB query, if the location value specified in either argument is malformed or invalid, then it will evaluate to **undefined** and the evaluated document to be skipped from the query results. If your query returns no results, run ST_ISVALIDDETAILED To debug why the spatail type is invalid.

Azure Cosmos DB also supports performing inverse queries, i.e. you can index Polygons or lines in Azure Cosmos DB, 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 providers stub methods `Distance()` and `Within()` for use within LINQ expressions. The DocumentDB LINQ provider translates these method calls to the equivalent SQL built-in function calls (ST_DISTANCE and ST_WITHIN respectively).

Here's an example of a LINQ query that finds all documents in the Azure Cosmos DB 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

Azure Cosmos DB for spatial indexing.

Indexing

As we described in the Schema Agnostic Indexing with Azure Cosmos DB paper, we designed Azure Cosmos DB's database engine to be truly schema agnostic and provide first class support for JSON. The write optimized database engine of Azure Cosmos DB 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 Azure Cosmos DB 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
>
> Azure Cosmos DB 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 Azure Cosmos DB, you can:

- Start coding with the Geospatial .NET code samples on GitHub
- Get hands on with geospatial querying at the Azure Cosmos DB Query Playground
- Learn more about Azure Cosmos DB Query
- Learn more about Azure Cosmos DB Indexing Policies

How does Azure Cosmos DB index data?

6/9/2017 • 19 min to read • Edit Online

By default, all Azure Cosmos DB data is indexed. And while many customers are happy to let Azure Cosmos DB automatically handle all aspects of indexing, Azure Cosmos DB also supports specifying a custom **indexing policy** for collections during creation. Indexing policies in Azure Cosmos DB 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 in Azure Cosmos DB, 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 Azure Cosmos DB 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 an Azure Cosmos DB 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**. Azure Cosmos DB supports three indexing modes which can be configured via the indexing policy on an Azure Cosmos DB 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, Azure Cosmos DB indexes all string properties within documents consistently with a Hash index, and numeric properties with a Range index.

Database indexing modes

Azure Cosmos DB supports three indexing modes which can be configured via the indexing policy on an Azure Cosmos DB collection – Consistent, Lazy and None.

Consistent: If an Azure Cosmos DB collection's policy is designated as "consistent", the queries on a given Azure Cosmos DB 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 an Azure Cosmos DB 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, an Azure Cosmos DB collection can be configured with lazy consistency; meaning queries are eventually consistent. The index is updated asynchronously when an Azure Cosmos DB 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 Azure Cosmos DB 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 an Azure Cosmos DB 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

Azure Cosmos DB 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 an Azure Cosmos DB collection using the .NET SDK with consistent indexing on all document insertions.

```
// Default collection creates a hash index for all string fields and a range index for all 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

Azure Cosmos DB models JSON documents and the index as trees, and allows you to tune to policies for paths within the tree. 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.

/prop/?	Index path required to serve queries like the following (with Hash or Range types respectively):
	SELECT FROM collection c WHERE c.prop = "value"
	SELECT FROM collection c WHERE c.prop > 5
	SELECT FROM collection c ORDER BY c.prop

PATH	DESCRIPTION/USE CASE

/prop/*	Index path for all paths under the specified label. Works with the following queries
	SELECT FROM collection c WHERE c.prop = "value"
	SELECT FROM collection c WHERE c.prop.subprop > 5
	SELECT FROM collection c WHERE c.prop.subprop.nextprop = "value"
	SELECT FROM collection c ORDER BY c.prop

/props/[]/?	Index path required to serve iteration and JOIN queries against arrays of scalars like ["a", "b", "c"]:
	SELECT tag FROM tag IN collection.props WHERE tag = "value"
	SELECT tag FROM collection c JOIN tag IN c.props WHERE tag > 5

/props/[]/subprop/?	Index path required to serve iteration and JOIN queries against arrays of objects like [{subprop: "a"}, {subprop: "b"}]:
	SELECT tag FROM tag IN collection.props WHERE tag.subprop = "value"
	SELECT tag FROM collection c JOIN tag IN c.props WHERE tag.subprop = "value"

/prop/subprop/?	Index path required to serve queries (with Hash or Range types respectively):
	SELECT FROM collection c WHERE c.prop.subprop = "value"
	SELECT FROM collection c WHERE c.prop.subprop > 5

> 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

Azure Cosmos DB supports Hash and Range index kinds for every path (that can 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. DataType 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). DataType can be String or Number.

Azure Cosmos DB 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. DataType can be Point, Polygon, or LineString.

> NOTE
>
> Azure Cosmos DB 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	Hash over /prop/? (or /) can be used to serve the following queries efficiently:
	SELECT FROM collection c WHERE c.prop = "value"
	Hash over /props/[]/? (or / or /props/) can be used to serve the following queries efficiently:
	SELECT tag FROM collection c JOIN tag IN c.props WHERE tag = 5

Range	Range over /prop/? (or /) can be used to serve the following queries efficiently:
	SELECT FROM collection c WHERE c.prop = "value"
	SELECT FROM collection c WHERE c.prop > 5
	SELECT FROM collection c ORDER BY c.prop

Spatial	Range over /prop/? (or /) can be used to serve the following queries efficiently: SELECT FROM collection c WHERE ST_DISTANCE(c.prop, {"type": "Point", "coordinates": [0.0, 10.0]}) < 40 SELECT FROM collection c WHERE ST_WITHIN(c.prop, {"type": "Polygon", ... }) --with indexing on points enabled SELECT FROM collection c WHERE ST_WITHIN({"type": "Point", ... }, c.prop) --with indexing on polygons enabled

By default, an error is returned for queries with range operators such as >= 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 Azure Cosmos DB 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 overriden.

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
>
> Azure Cosmos DB 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.

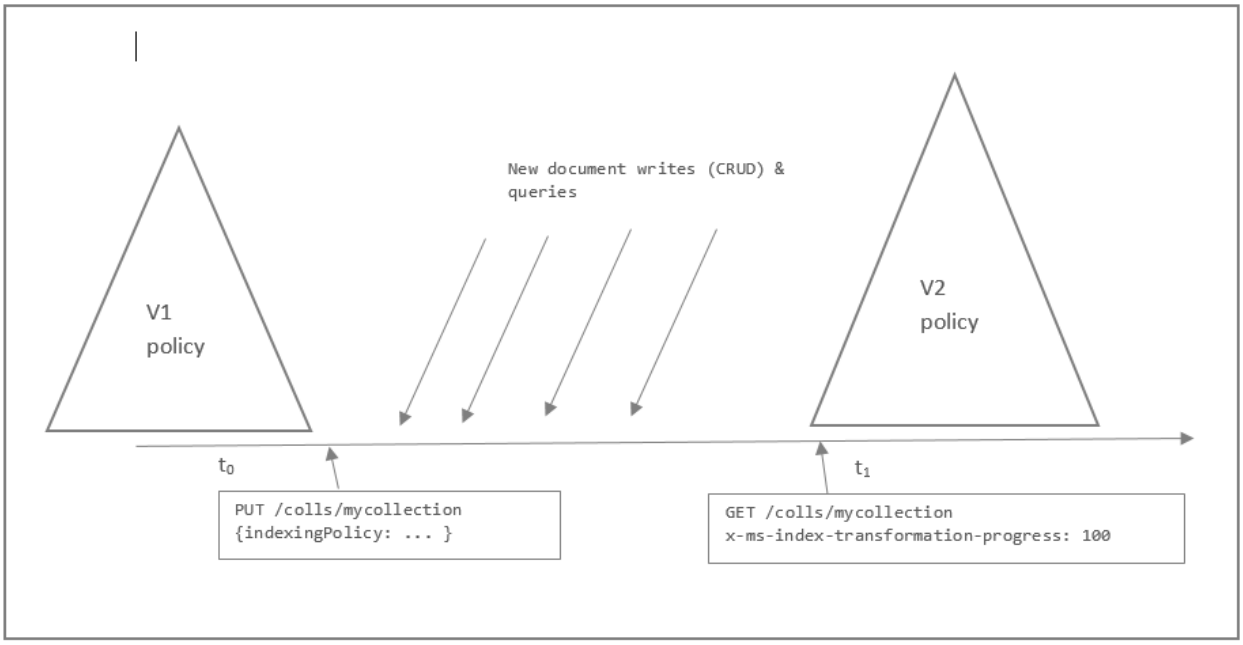
For example, the following sample shows how to include a document explicitly using the DocumentDB API .NET SDK and the RequestOptions.IndexingDirective property.

```
// 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

Azure Cosmos DB allows you to make changes to the indexing policy of a collection on the fly. A change in indexing policy on an Azure Cosmos DB 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. Azure Cosmos DB 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 Azure Cosmos DB 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 Azure Cosmos DB 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 of 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 Azure Cosmos DB 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 Azure Cosmos DB 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 >= 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 Azure Cosmos DB's query language.

1. DocumentDB API .NET Index Management code samples
2. DocumentDB API REST Collection Operations
3. Query with SQL

Azure Cosmos DB as a key value store – Cost overview

6/6/2017 • 4 min to read • Edit Online

Azure Cosmos DB is a globally distributed, multi-model database service for building highly available, large scale applications easily. By default, Azure Cosmos DB automatically indexes all the data it ingests, efficiently. This enables fast and consistent SQL (and JavaScript) queries on any kind of data.

This article describes the cost of Azure Cosmos DB for simple write and read operations when it's used as a key/value store. Write operations include inserts, replaces, deletes, and upserts of documents. Besides guaranteeing 99.99% high availability, Azure Cosmos DB offers guaranteed <10 ms latency for reads and <15 ms latency for the (indexed) writes respectively, at the 99th percentile.

Why we use Request Units (RUs)

Azure Cosmos DB performance is based on the amount of provisioned Request Units (RU) for the partition. The provisioning is at a second granularity and is purchased in RUs/sec (not to be confused with the hourly billing). RUs should be considered as a currency that simplifies the provisioning of required throughput for the application. Our customers do not have to think of differentiating between read and write capacity units. The single currency model of RUs creates efficiencies to share the provisioned capacity between reads and writes. This provisioned capacity model enables the service to provide a predictable and consistent throughput, guaranteed low latency, and high availability. Finally, we use RU to model throughput but each provisioned RU has also a defined amount of resources (Memory, Core). RU/sec is not only IOPS.

As a globally distributed database system, Azure Cosmos DB is the only Azure service that provides an SLA on latency, throughput, and consistency in addition to high availability. The throughput you provision is applied to each of the regions associated with your Azure Cosmos DB database account. For reads, Azure Cosmos DB offers multiple, well-defined consistency levels for you to choose from. Azure Cosmos DB is a globally distributed, multi-model database service for building highly available, large scale, globally distributed applications easily. By default, Cosmos DB automatically indexes all the data it ingests, efficiently. This enables fast and consistent SQL (and JavaScript) queries on any kind of data.

This article describes the cost of Cosmos DB for simple write and read operations when it's used as a key/value store. Write operations include inserts, replaces, deletes, and upserts of documents. Besides guaranteeing 99.99% high availability, Cosmos DB offers guaranteed <10 ms latency for reads and <15 ms latency for the (indexed) writes respectively, at the 99th percentile.

Why we use Request Units (RUs)

Cosmos DB performance is based on the amount of provisioned Request Units (RU) for the partition. The provisioning is at a second granularity and is purchased in RUs/sec and RUs/min (not to be confused with the hourly billing). RUs should be considered as a currency that simplifies the provisioning of required throughput for the application. Our customers do not have to think of differentiating between read and write capacity units. The single currency model of RUs creates efficiencies to share the provisioned capacity between reads and writes. This provisioned capacity model enables the service to provide a predictable and consistent throughput, guaranteed low latency, and high availability. Finally, we use RU to model throughput but each provisioned RU has also a defined amount of resources (Memory, Core). RU/sec is not only IOPS.

As a globally distributed database system, Cosmos DB is the only Azure service that provides an SLA on latency, throughput, and consistency in addition to high availability. The throughput you provision is applied to each of the regions associated with your Cosmos DB database account. For reads, Cosmos DB offers multiple, well-defined consistency levels for you to choose from.

The following table shows the number of RUs required to perform read and write transactions based on document size of 1KB and 100KBs.

ITEM SIZE	1 READ	1 WRITE
1 KB	1 RU	5 RUs
100 KB	10 RUs	50 RUs

Cost of Reads and Writes

If you provision 1,000 RU/sec, this amounts to 3.6m RU/hour and will cost $0.08 for the hour (in the US and Europe). For a 1KB size document, this means that you can consume 3.6m reads or 0.72m writes (3.6mRU / 5) using your provisioned throughput. Normalized to million reads and writes, the cost would be $0.022 /m reads ($0.08 / 3.6) and $0.111/m writes ($0.08 / 0.72). The cost per million becomes minimal as shown in the table below.

ITEM SIZE	1M READ	1M WRITE
1 KB	$0.022	$0.111
100 KB	$0.222	$1.111

Most of the basic blob or object stores services charge $0.40 per million read transaction and $5 per million write transaction. If used optimally, Cosmos DB can be up to 98% cheaper than these other solutions (for 1KB transactions).

Next steps

Stay tuned for new articles on optimizing Cosmos DB resource provisioning. In the meantime, feel free to use our RU calculator.

Expire data in Azure Cosmos DB collections automatically with time to live

5/30/2017 • 7 min to read • Edit Online

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 Cosmos DB 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, Cosmos DB will automatically remove documents that exist after that period of time, in seconds, since they were last modified.

Time to live in Cosmos DB 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 (`ttl` + `_ts` >= 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 in 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 Cosmos DB 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 Cosmos DB service should monitor this collection for documents that have overridden this default.

```
DocumentCollection collectionDefinition = new DocumentCollection();
collectionDefinition.Id = "orders";
collectionDefinition.PartitionKey.Paths.Add("/customerId");
collectionDefinition.DefaultTimeToLive =-1; //never expire by default

DocumentCollection ttlEnabledCollection = await client.CreateDocumentCollectionAsync(
    UriFactory.CreateDatabaseUri(databaseName),
    collectionDefinition,
    new RequestOptions { OfferThroughput = 20000 });
```

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.

```
DocumentCollection collectionDefinition = new DocumentCollection();
collectionDefinition.Id = "orders";
collectionDefinition.PartitionKey.Paths.Add("/customerId");
collectionDefinition.DefaultTimeToLive = 90 * 60 * 60 * 24; // expire all documents after 90 days

DocumentCollection ttlEnabledCollection = await client.CreateDocumentCollectionAsync(
    "/dbs/salesdb",
    collectionDefinition,
    new RequestOptions { OfferThroughput = 20000 });
```

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`).
- 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.

Here's a snippet showing how to set the TTL expiration on a document:

```
// Include a property that serializes to "ttl" in JSON
public class SalesOrder
{
    [JsonProperty(PropertyName = "id")]
    public string Id { get; set; }

    [JsonProperty(PropertyName="cid")]
    public string CustomerId { get; set; }

    // used to set expiration policy
    [JsonProperty(PropertyName = "ttl", NullValueHandling = NullValueHandling.Ignore)]
    public int? TimeToLive { get; set; }

    //...
}

// Set the value to the expiration in seconds
SalesOrder salesOrder = new SalesOrder
{
    Id = "SO05",
    CustomerId = "CO18009186470",
    TimeToLive = 60 * 60 * 24 * 30;  // Expire sales orders in 30 days
};
```

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.

```
response = await client.ReadDocumentAsync(
    "/dbs/salesdb/colls/orders/docs/SO05"),
    new RequestOptions { PartitionKey = new PartitionKey("CO18009186470") });

Document readDocument = response.Resource;
readDocument.TimeToLive = 60 * 30 * 30; // update time to live

response = await client.ReplaceDocumentAsync(salesOrder);
```

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.

```
response = await client.ReadDocumentAsync(
    "/dbs/salesdb/colls/orders/docs/SO05"),
    new RequestOptions { PartitionKey = new PartitionKey("CO18009186470") });

Document readDocument = response.Resource;
readDocument.TimeToLive = null; // inherit the default TTL of the collection

response = await client.ReplaceDocumentAsync(salesOrder);
```

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.

```
DocumentCollection collection = await client.ReadDocumentCollectionAsync("/dbs/salesdb/colls/orders");

// Disable TTL
collection.DefaultTimeToLive = null;

await client.ReplaceDocumentCollectionAsync(collection);
```

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 Cosmos DB.

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 Cosmos DB, refer to the service *documentation* page.

Automatic online backup and restore with Azure Cosmos DB

5/30/2017 • 3 min to read • Edit Online

Azure Cosmos DB automatically takes backups of all your data at regular intervals. The automatic backups are taken without affecting the performance or availability of your 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 Comos DB container and later require data recovery or a disaster recovery solution.

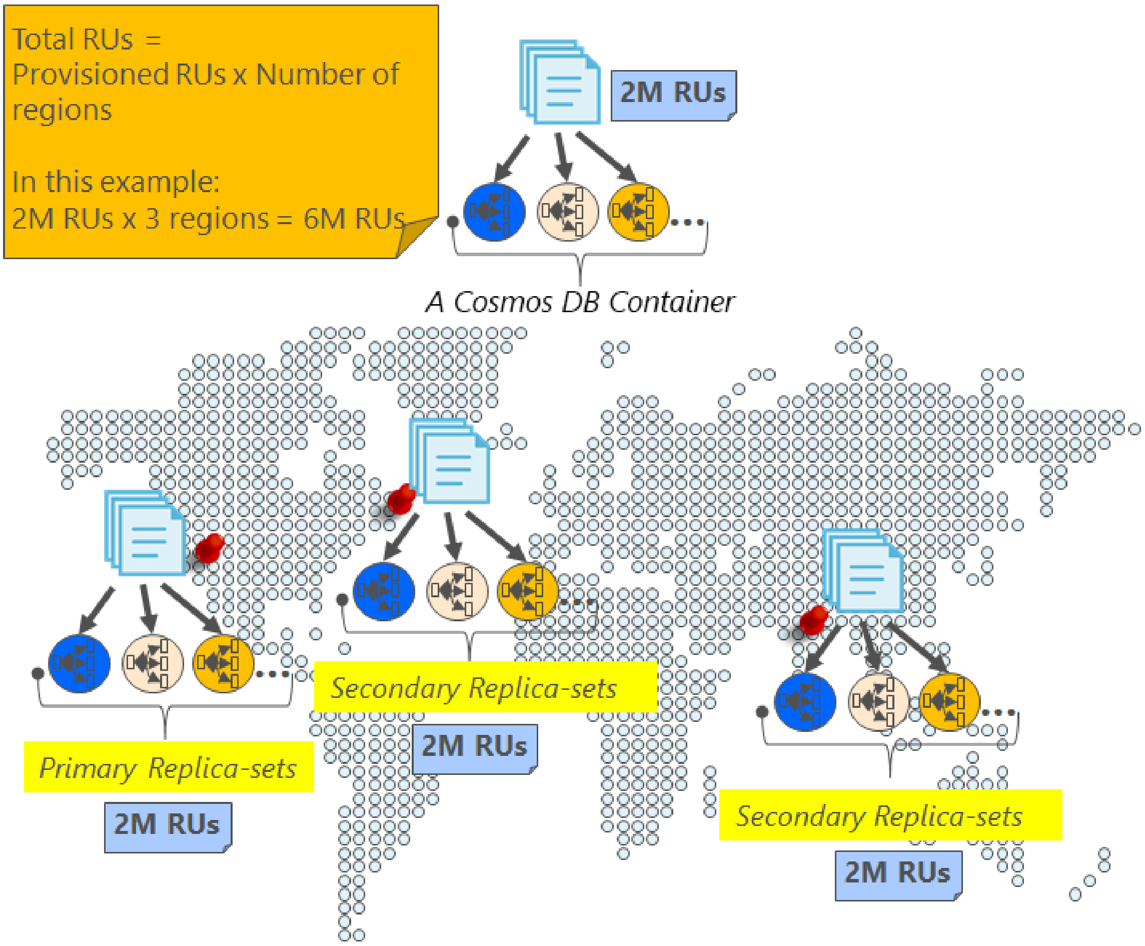
This article starts with a quick recap of the data redundancy and availability in Cosmos DB, and then discusses backups.

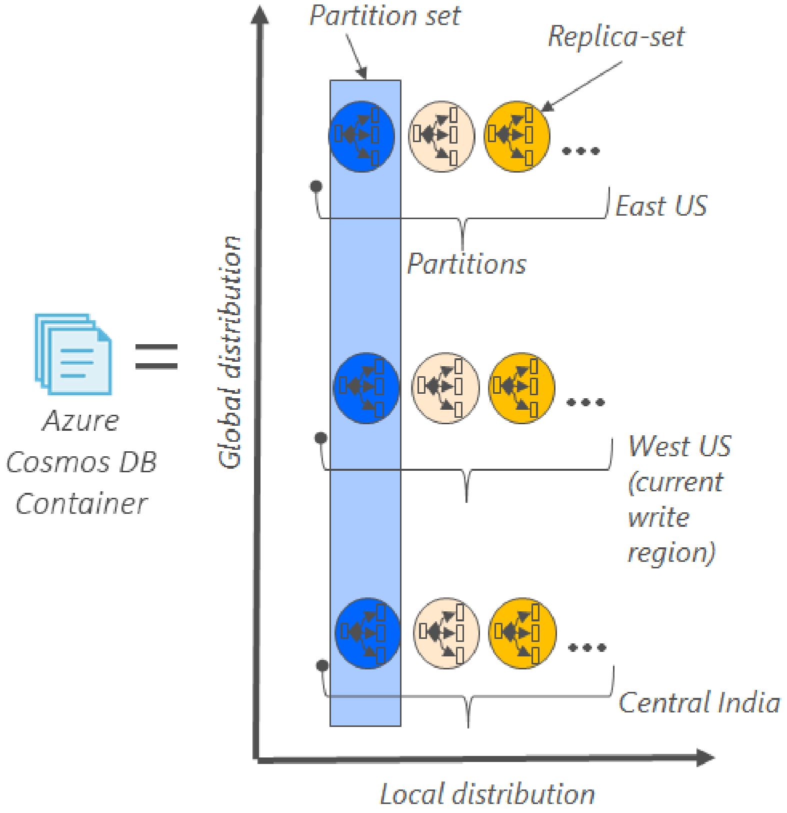
High availability with Cosmos DB - a recap

Cosmos DB 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 Cosmos DB 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 Cosmos DB 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 Cosmos DB container 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 Cosmos DB 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 Cosmos DB.





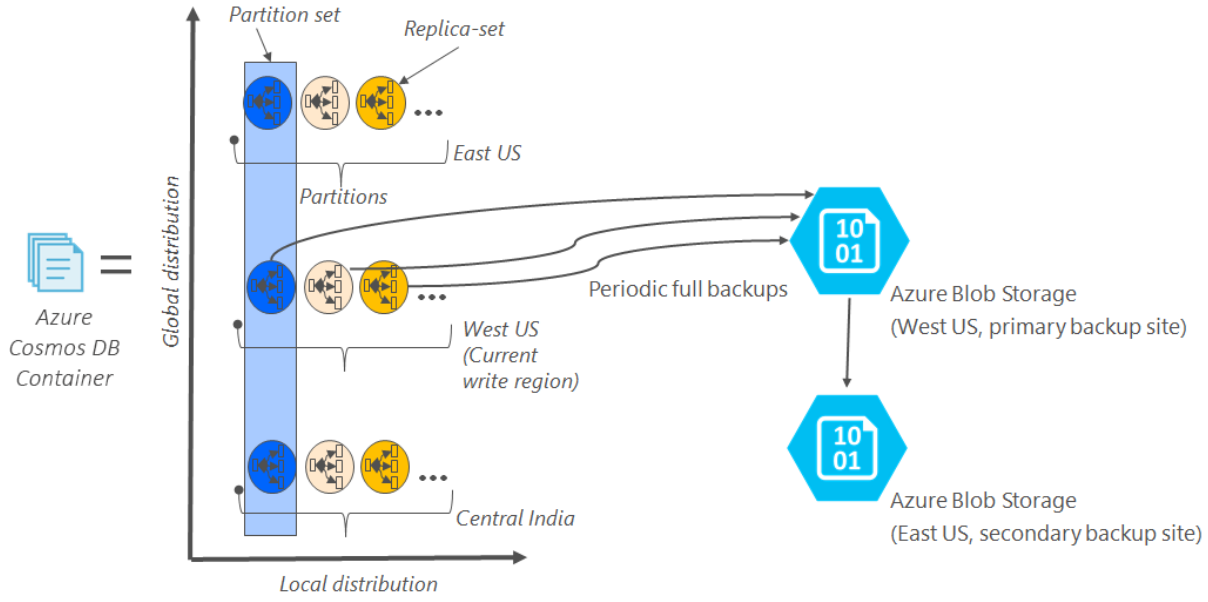
Full, automatic, online backups

Oops, I deleted my container or database! With Cosmos DB, 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 and latest 2 backups are stored at all times. If the data is accidently dropped or corrupted, please contact Azure support within 8 hours.

The backups are taken without affecting the performance or availability of your database operations. Cosmos DB takes the backup in the background without consuming your provisioned RUs or affecting the performance and without affecting the availability of your database.

Unlike your data that is stored inside Cosmos DB, 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 Cosmos DB 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 Cosmos DB container (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 Cosmos DB 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 container or account is deleted, Cosmos DB 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, Cosmos DB 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 database in multiple data centers, see distribute your data globally with Cosmos DB.

To file contact Azure Support, file a ticket from the Azure portal.

Automatic regional failover for business continuity in Azure Cosmos DB

5/30/2017 • 6 min to read • Edit Online

Azure Cosmos DB 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. Cosmos DB accounts offer high availability, single digit ms latencies, well-defined consistency levels, transparent regional failover with multi-homing APIs, and the ability to elastically scale throughput and storage across the globe.

Cosmos DB supports both explicit and policy driven failovers that allow you to control the end-to-end system behavior in the event of failures. In this article, we look at:

- How do manual failovers work in Cosmos DB?
- How do automatic failovers work in Cosmos DB and what happens when a data center goes down?
- How can you use manual failovers in application architectures?

You can also learn about regional failovers in this Azure Friday video with Scott Hanselman and Principal Engineering Manager Karthik Raman.

Configuring multi-region applications

Before we dive into failover modes, we look at how you can configure an application to take advantage of multi-region availability and be resilient in the face of regional failovers.

- First, deploy your application in multiple regions
- To ensure low latency access from every region your application is deployed, configure the corresponding preferred regions list for each region via one of the supported SDKs.

The following snippet shows how to initialize a multi-region application. Here, the Azure Cosmos DB account `contoso.documents.azure.com` is configured with two regions - West US and North Europe.

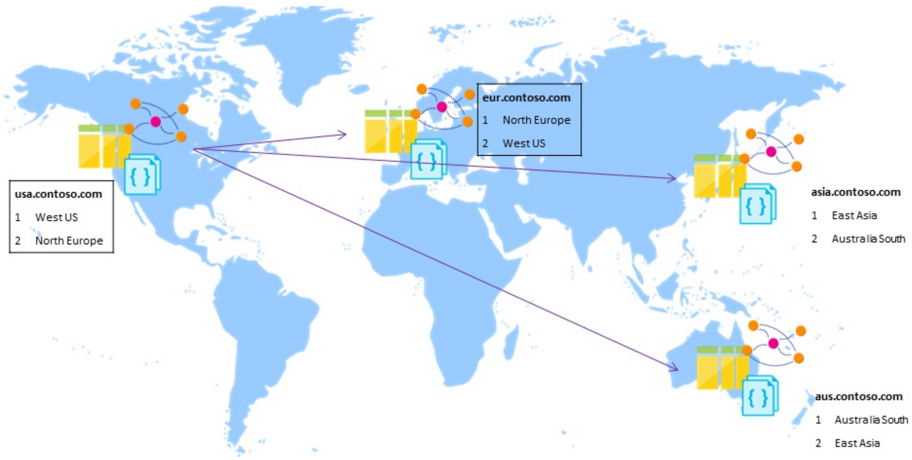
- The application is deployed in the West US region (using Azure App Services for example)
- Configured with `West US` as the first preferred region for low latency reads
- Configured with `North Europe` as the second preferred region (for high availability during regional failures)

In the DocumentDB API, this configuration looks like the following snippet:

```
ConnectionPolicy usConnectionPolicy = new ConnectionPolicy
{
   ConnectionMode = ConnectionMode.Direct,
   ConnectionProtocol = Protocol.Tcp
};

usConnectionPolicy.PreferredLocations.Add(LocationNames.WestUS);
usConnectionPolicy.PreferredLocations.Add(LocationNames.NorthEurope);

DocumentClient usClient = new DocumentClient(
   new Uri("https://contosodb.documents.azure.com"),
   "memf7qfF89n6KL9vcb7rIQl6tfgZsRt5gY5dh3BIjesarJanYIcg2Edn9uPOUIVwgkAugOb2zUdCR2h0PTtMrA==",
   usConnectionPolicy);
```

The application is also deployed in the North Europe region with the order of preferred regions reversed. That is, the North Europe region is specified first for low latency reads. Then, the West US region is specified as the second preferred region for high availability during regional failures.

The following architecture diagram shows a multi-region application deployment where Cosmos DB and the application are configured to be available in four Azure geographic regions.



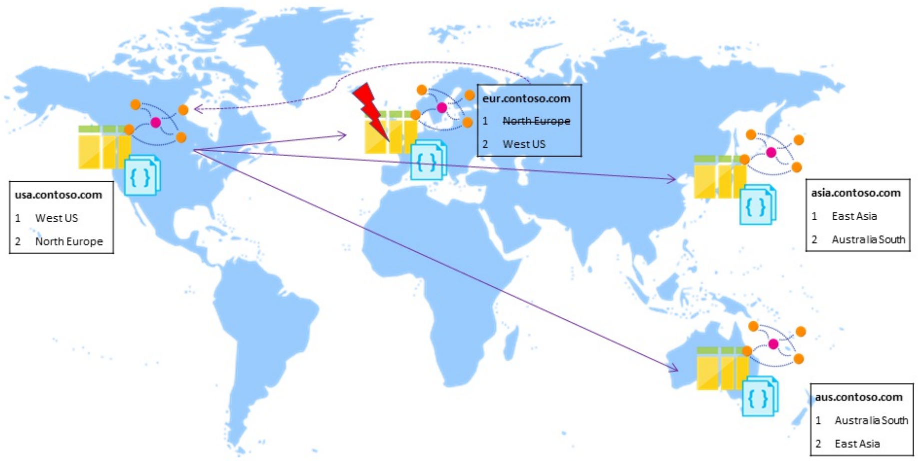
Now, let's look at how the Cosmos DB service handles regional failures via automatic failovers.

Automatic Failovers

In the rare event of an Azure regional outage or data center outage, Cosmos DB automatically triggers failovers of all Cosmos DB accounts with a presence in the affected region.

What happens if a read region has an outage?

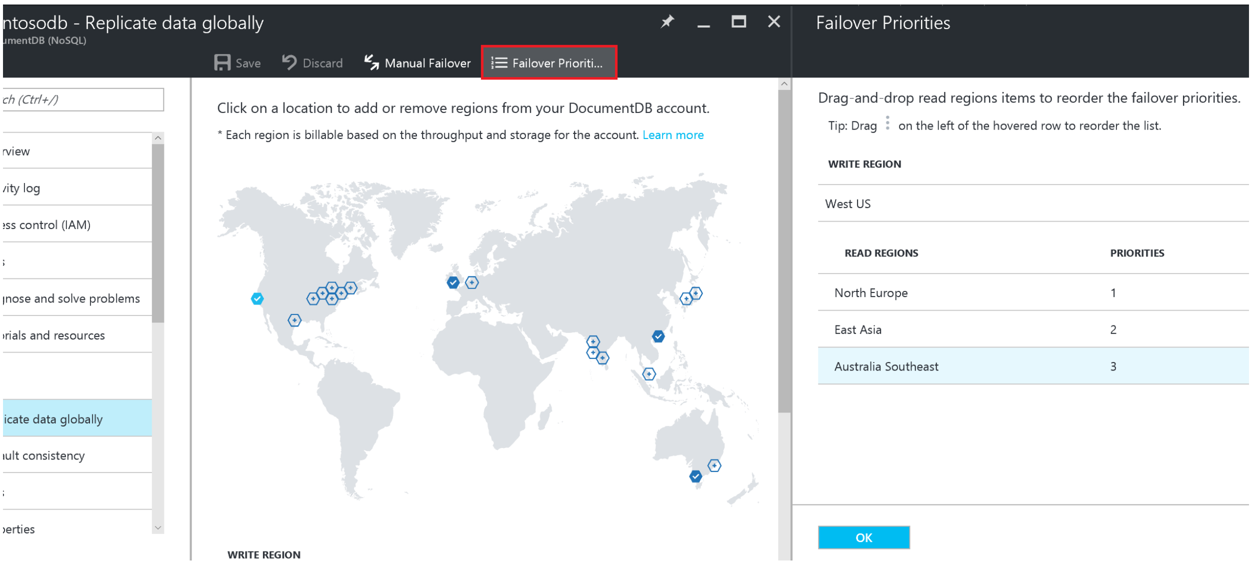
Cosmos DB accounts with a read region in one of the affected regions are automatically disconnected from their write region and marked offline. The Cosmos DB SDKs implement a regional discovery protocol that allows them to automatically detect when a region is available and redirect read calls to the next available region in the preferred region list. If none of the regions in the preferred region list is available, calls automatically fall back to the current write region. No changes are required in your application code to handle regional failovers. During this entire process, consistency guarantees continue to be honored by Cosmos DB.



Once the affected region recovers from the outage, all the affected Cosmos DB accounts in the region are automatically recovered by the service. Cosmos DB accounts that had a read region in the affected region will then automatically sync with current write region and turn online. The Cosmos DB SDKs discover the availability of the new region and evaluate whether the region should be selected as the current read region based on the preferred region list configured by the application. Subsequent reads are redirected to the recovered region without requiring any changes to your application code.

What happens if a write region has an outage?

If the affected region is the current write region for a given Cosmos DB account, then the region will be automatically marked as offline. Then, an alternative region is promoted as the write region each affected Cosmos DB account. You can fully control the region selection order for your Cosmos DB accounts via the Azure portal or programmatically.



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

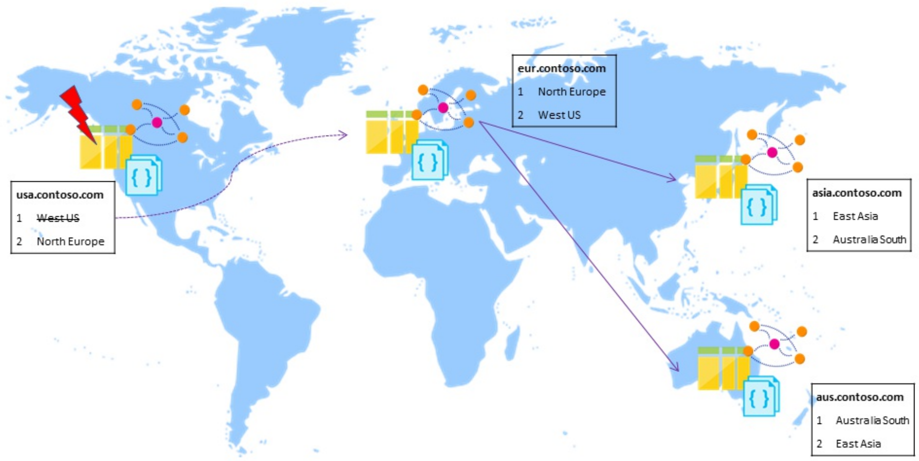
Drag-and-drop read regions items to reorder the failover priorities.

Tip: Drag ⋮ on the left of the hovered row to reorder the list.

WRITE REGION
West US

READ REGIONS	PRIORITIES
North Europe	1
East Asia	2
Australia Southeast	3

During automatic failovers, Cosmos DB automatically chooses the next write region for a given Cosmos DB account based on the specified priority order.



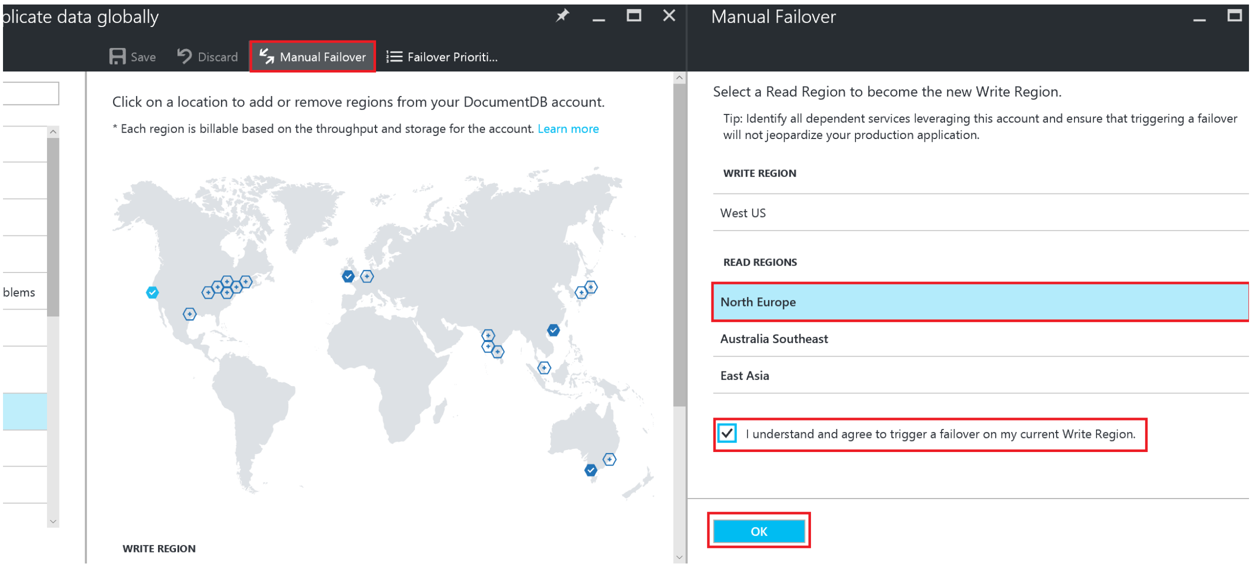
Once the affected region recovers from the outage, all the affected Cosmos DB accounts in the region are automatically recovered by the service.

- Cosmos DB accounts with their previous write region in the affected region will stay in an offline mode with read availability even after the recovery of the region.
- You can query this region to compute any unreplicated writes during the outage by comparing with the data available in the current write region. Based on the needs of your application, you can perform merge and/or conflict resolution and write the final set of changes back to the current write region.
- Once you've completed merging changes, you can bring the affected region back online by removing and readding the region to your Cosmos DB account. Once the region is added back, you can configure it back as the write region by performing a manual failover via the Azure portal or programmatically.

Manual Failovers

In addition to automatic failovers, the current write region of a given Cosmos DB account can be manually changed dynamically to one of the existing read regions. Manual failovers can be initiated via the Azure portal or programmatically.

Manual failovers ensure **zero data loss** and **zero availability** loss and gracefully transfer write status from the old write region to the new one for the specified Cosmos DB account. Like in automatic failovers, the Cosmos DB SDK automatically handles write region changes during manual failovers and ensures that calls are automatically redirected to the new write region. No code or configuration changes are required in your application to manage failovers.



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

Select a Read Region to become the new Write Region.

Tip: Identify all dependent services leveraging this account and ensure that triggering a failover will not jeopardize your production application.

WRITE REGION

West US

READ REGIONS

North Europe

Australia Southeast

East Asia

☑ I understand and agree to trigger a failover on my current Write Region.

Some of the common scenarios where manual failover can be useful are:

Follow the clock model: If your applications have predictable traffic patterns based on the time of the day, you can periodically change the write status to the most active geographic region based on time of the day.

Service update: Certain globally distributed application deployment may involve rerouting traffic to different region via traffic manager during their planned service update. Such application deployment now can use manual failover to keep the write status to the region where there is going to be active traffic during the service update window.

Business Continuity and Disaster Recovery (BCDR) and High Availability and Disaster Recovery (HADR) drills: Most enterprise applications include business continuity tests as part of their development and release process. BCDR and HADR testing is often an important step in compliance certifications and guaranteeing service availability in the case of regional outages. You can test the BCDR readiness of your applications that use Cosmos DB for storage by triggering a manual failover of your Cosmos DB account and/or adding and removing a region dynamically.

In this article, we reviewed how manual and automatic failovers work in Cosmos DB, and how you can configure your Cosmos DB accounts and applications to be globally available. By using Cosmos DB's global replication support, you can improve end-to-end latency and ensure that they are highly available even in the event of region failures.

Next Steps

- Learn about how Cosmos DB supports global distribution
- Learn about global consistency with Azure Cosmos DB
- Develop with multiple regions using Azure Cosmos DB's DocumentDB SDK
- Learn how to build Multi-region writer architectures with Azure DocumentDB

Set throughput for Azure Cosmos DB containers

6/12/2017 • 1 min to read • Edit Online

You can set throughput for your Azure Cosmos DB containers in the Azure portal or by using the client SDKs.

The following table lists the throughput available for containers:

	Single Partition Container	**Partitioned Container**
Minimum Throughput	400 request units per second	2,500 request units per second
Maximum Throughput	10,000 request units per second	Unlimited

To set the throughput by using the Azure portal

1. In a new window, open the Azure portal.
2. On the left bar, click **Azure Cosmos DB**, or click **More Services** at the bottom, then scroll to **Databases**, and then click **Azure Cosmos DB**.
3. Select your Cosmos DB account.
4. In the new window, click **Data Explorer (Preview)** in the navigation menu.
5. In the new window, expand your database and container and then click **Scale & Settings**.
6. In the new window, type the new throughput value in the **Throughput** box, and then click **Save**.

To set the throughput by using the DocumentDB API for .NET

```
//Fetch the resource to be updated
Offer offer = client.CreateOfferQuery()
    .Where(r => r.ResourceLink == collection.SelfLink)
    .AsEnumerable()
    .SingleOrDefault();

// Set the throughput to the new value, for example 12,000 request units per second
offer = new OfferV2(offer, 12000);

//Now persist these changes to the database by replacing the original resource
await client.ReplaceOfferAsync(offer);
```

Throughput FAQ

Can I set my throughput to less than 400 RU/s?

400 RU/s is the minimum throughput available on Cosmos DB single partition collections (2500 RU/s is the minimum for partitioned collections). Request units are set in 100 RU/s intervals, but throughput cannot be set to 100 RU/s or any value smaller than 400 RU/s. If you're looking for a cost effective method to develop and test Cosmos DB, you can use the free Azure Cosmos DB Emulator, which you can deploy locally at no cost.

How do I set througput using the MongoDB API?

There's no MongoDB API extension to set throughput. The recommendation is to use the DocumentDB API, as shown in To set the throughput by using the DocumentDB API for .NET.

Next steps

To learn more about provisioning and going planet-scale with Cosmos DB, see Partitioning and scaling with Cosmos DB.

Monitor Azure Cosmos DB requests, usage, and storage

5/30/2017 • 4 min to read • Edit Online

You can monitor your Azure Cosmos DB accounts in the Azure portal. For each Azure Cosmos DB 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, the new Metrics blade, or in Azure Monitor.

View performance metrics on the Metrics blade

1. In the Azure portal, click **More Services**, scroll to **Databases**, click **Azure Cosmos DB**, and then click the name of the Azure Cosmos DB account for which you would like to view performance metrics.
2. In the resource menu, under **Monitoring**, 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 Azure Cosmos DB SLAs.

View performance metrics by using Azure Monitoring

1. In the Azure portal, click **Monitor** on the Jumpbar.
2. In the resource menu, click **Metrics**.
3. In the **Monitor - Metrics** window, in the **esource group** drop-down menu, select the resource group associated with the Azure Cosmos DB account that you'd like to monitor.
4. In the **Resource** drop-down menu, select the database account to monitor.
5. In the list of **Available metrics**, select the metrics to display. Use the CTRL button to multi-select.

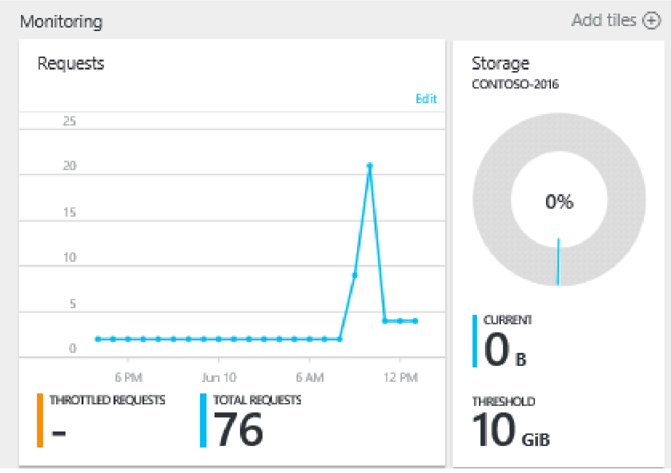
   Your metrics are displayed on in the **Plot** window.

View performance metrics on the account blade

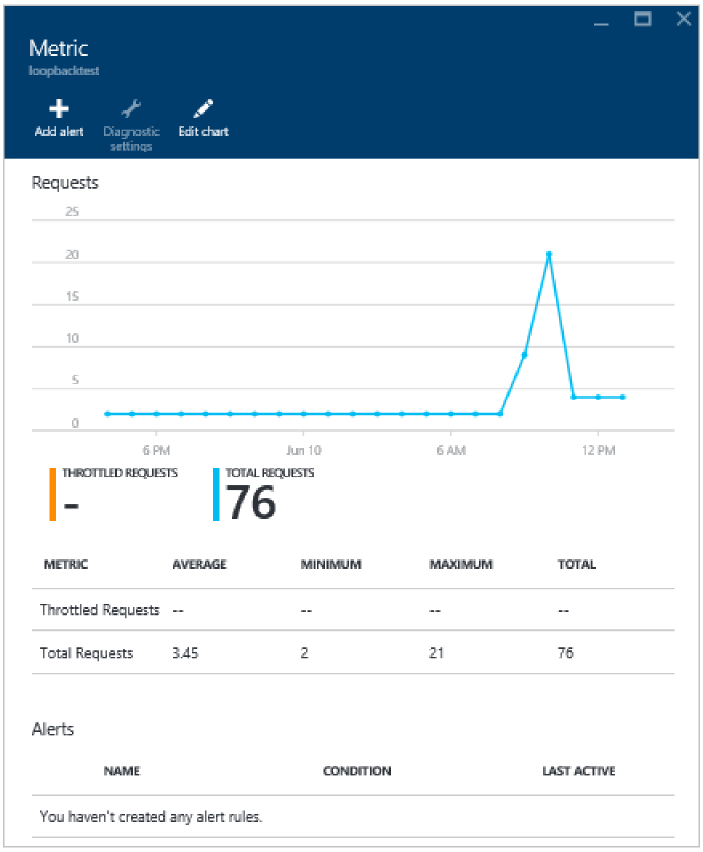
1. In the Azure portal, click **More Services**, scroll to **Databases**, click **Azure Cosmos DB**, and then click the name of the Azure Cosmos DB account for which you would like to view performance metrics.
2. The **Monitoring** lens displays the following tiles by default:

   - 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 **Usage Quota** 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 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).



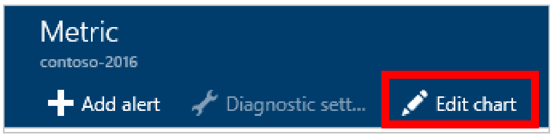
METRIC	AVERAGE	MINIMUM	MAXIMUM	TOTAL
Throttled Requests	--	--	--	--
Total Requests	3.45	2	21	76

Alerts

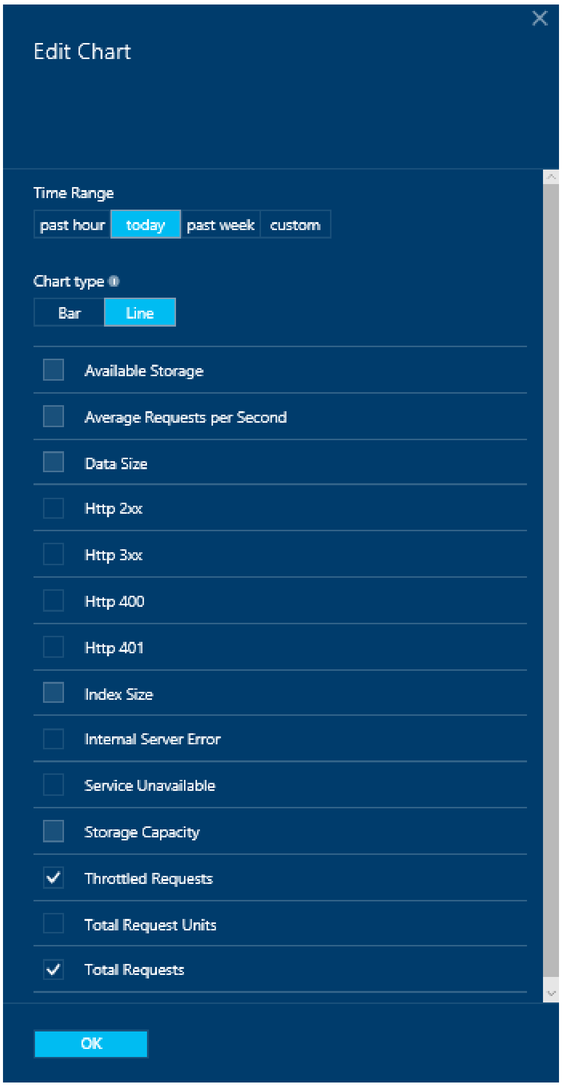
NAME	CONDITION	LAST ACTIVE
You haven't created any alert rules.		

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

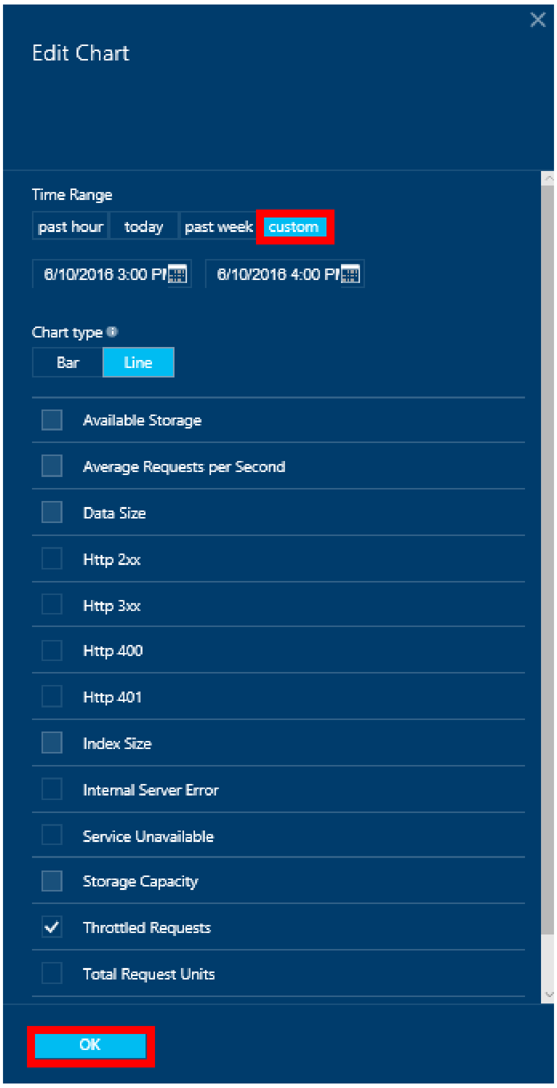
Chart type

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.



Edit Chart

Time Range

past hour | today | past week | custom

6/10/2016 3:00 PM | 6/10/2016 4:00 PM

Chart type

Bar | Line

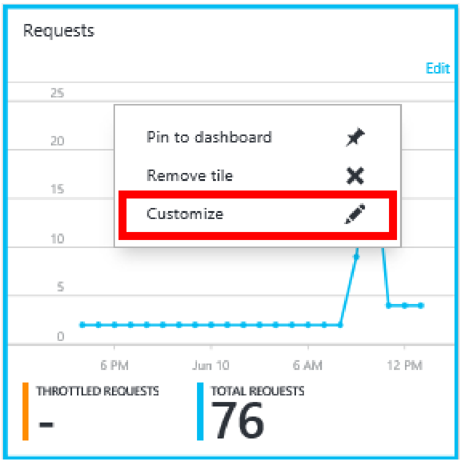
- □ Available Storage
- □ Average Requests per Second
- □ Data Size
- □ Http 2xx
- □ Http 3xx
- □ Http 400
- □ Http 401
- □ Index Size
- □ Internal Server Error
- □ Service Unavailable
- □ Storage Capacity
- ☑ Throttled Requests
- □ Total Request Units

OK

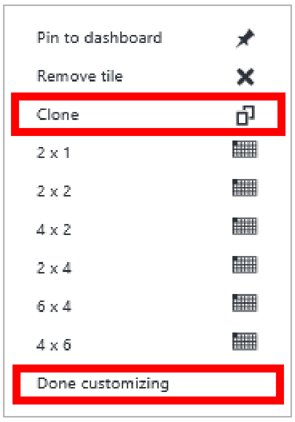
Create side-by-side charts in the portal

The Azure Portal allows you to create side-by-side metric charts.

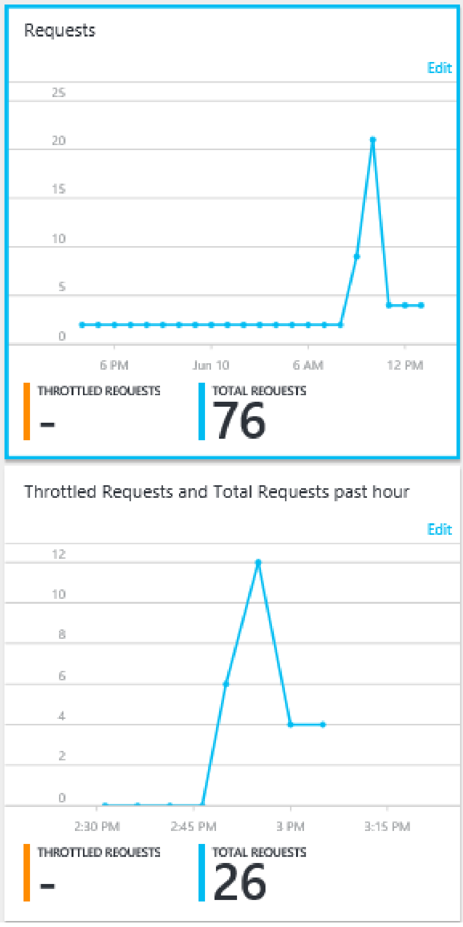
1. First, right-click on the chart you want to copy and select **Customize**.



2. Click **Clone** on the menu to copy the part and then click **Done customizing**.

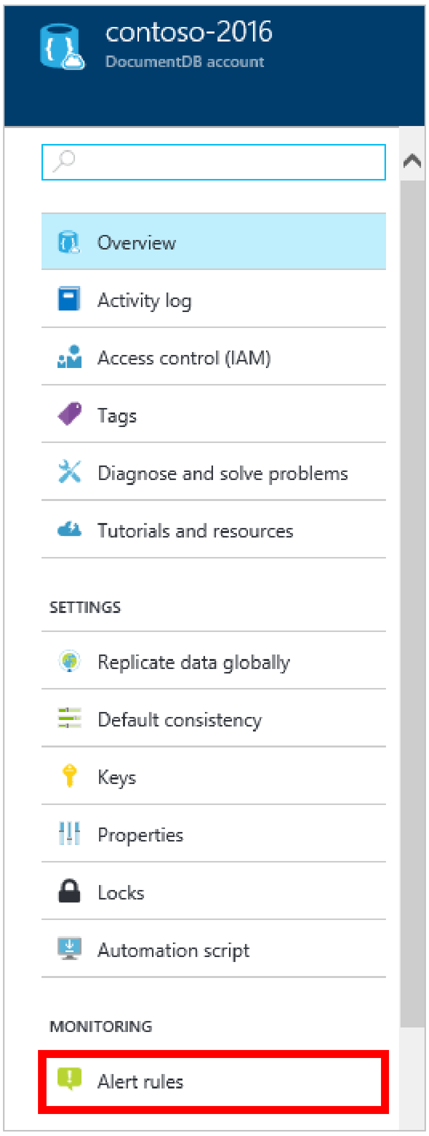


You may now treat this part as any other metric part, customizing the metrics and time range displayed in the part. By doing this, you can see two different metrics chart side-by-side at the same time.

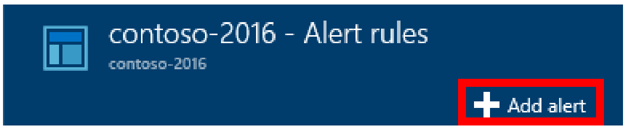


Set up alerts in the portal

1. In the Azure portal, click **More Services**, click **Azure Cosmos DB**, and then click the name of the Azure Cosmos DB 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.

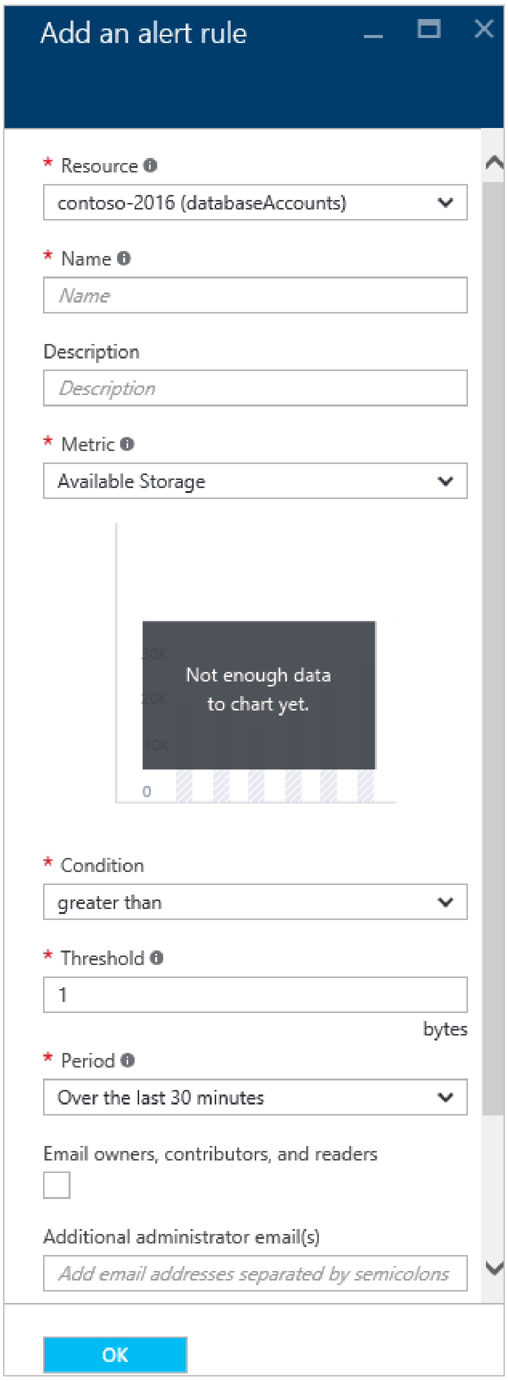


3. In the **Alert rules** blade, click **Add alert**.



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

* 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 Azure Cosmos DB programatically

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/database
Accounts/{DocumentDBAccountName}/metricDefinitions?api-version=2015-04-08
```

Queries to retrieve individual metrics use the following format:

```
https://management.azure.com/subscriptions/{SubecriptionId}/resourceGroups/{ResourceGroup}/providers/Microsoft.DocumentDb/database
Accounts/{DocumentDBAccountName}/metrics?api-version=2015-04-
08&$filter=%28name.value%20eq%20%27Total%20Requests%27%29%20and%20timeGrain%20eq%20duration%27PT5M%27%20and%20start
Time%20eq%202016-06-03T03%3A26%3A00.0000000Z%20and%20endTime%20eq%202016-06-10T03%3A26%3A00.0000000Z
```

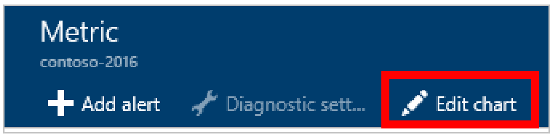
For more information, see Retrieving Resource Metrics via the Azure Monitor REST API. Note that "Azure Inights" was renamed "Azure Monitor". This blog entry refers to the older name.

Troubleshooting

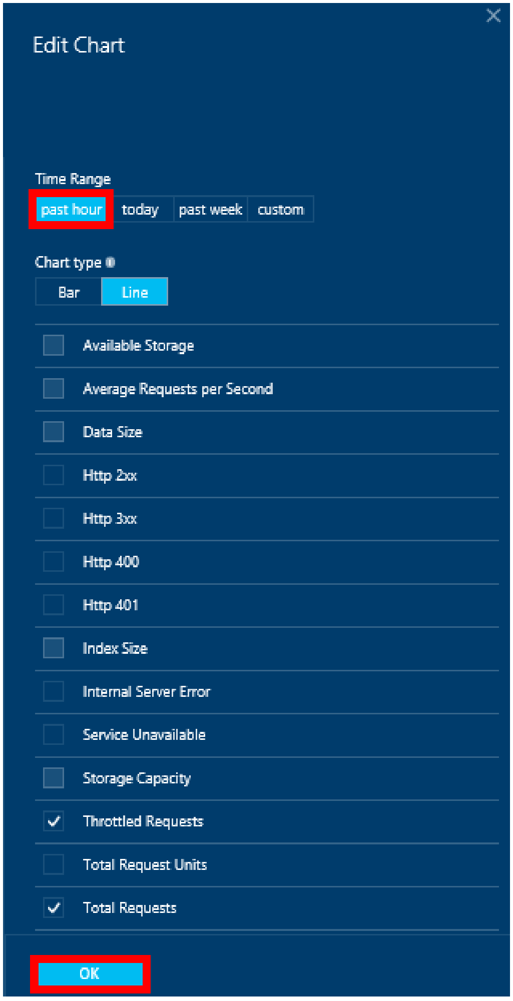
If your monitoring tiles display the **No data available** message, and you recently made requests or added data to the database, you can edit the tile to reflect the recent usage.

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**.

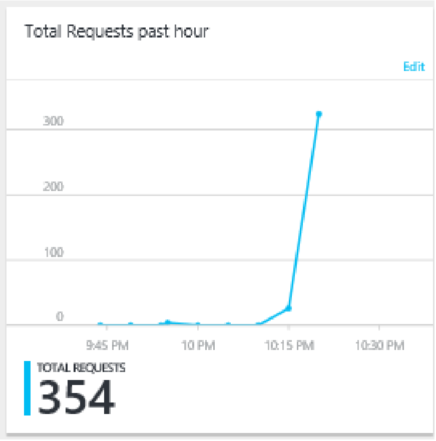

Edit Chart

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

3. Your tile should now refresh showing your current data and usage.



Next steps

To learn more about Azure Cosmos DB capacity planning, see the Azure Cosmos DB capacity planner calculator.

How to manage an Azure Cosmos DB account

5/30/2017 • 4 min to read • Edit Online

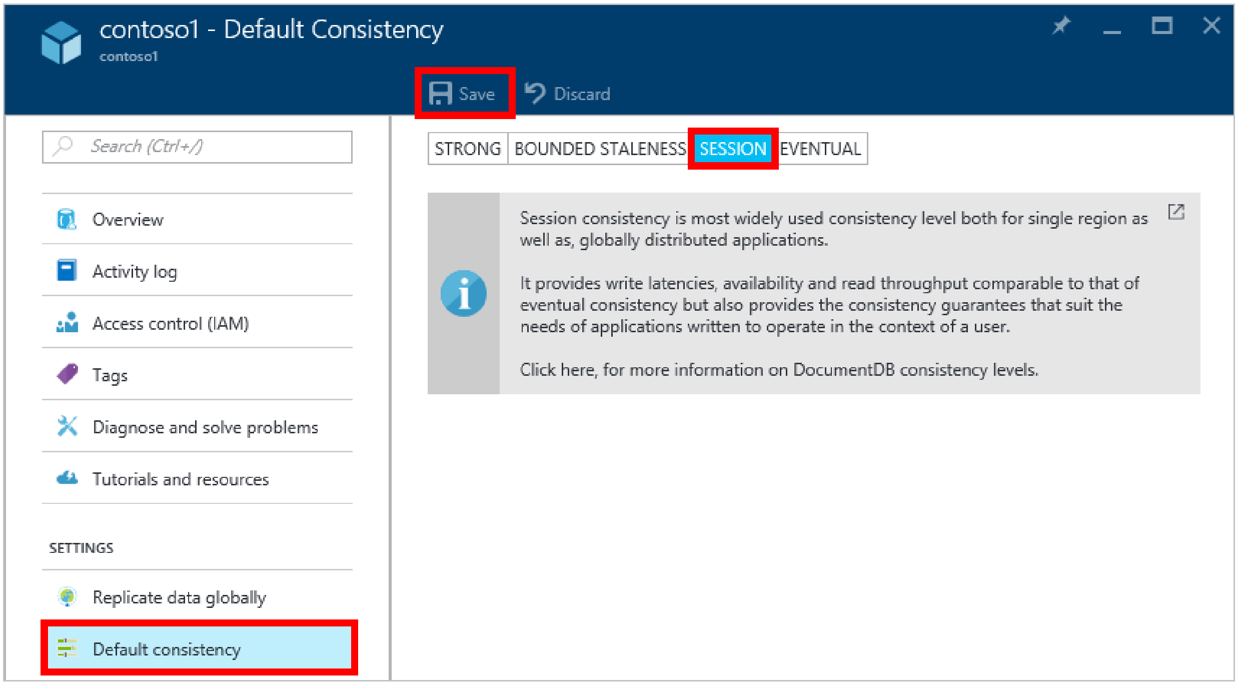
Learn how to set global consistency, work with keys, and delete an Azure Cosmos DB account in the Azure portal.

Manage Azure Cosmos DB consistency settings

Selecting the right consistency level depends on the semantics of your application. You should familiarize yourself with the available consistency levels in Azure Cosmos DB by reading Using consistency levels to maximize availability and performance in Azure Cosmos DB. Azure Cosmos DB 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 an Azure Cosmos DB account

1. In the Azure portal, access your Azure Cosmos DB account.
2. In the account blade, click **Default consistency**.
3. In the **Default Consistency** blade, select the new consistency level and click **Save**.



Session consistency is most widely used consistency level both for single region as well as, globally distributed applications.

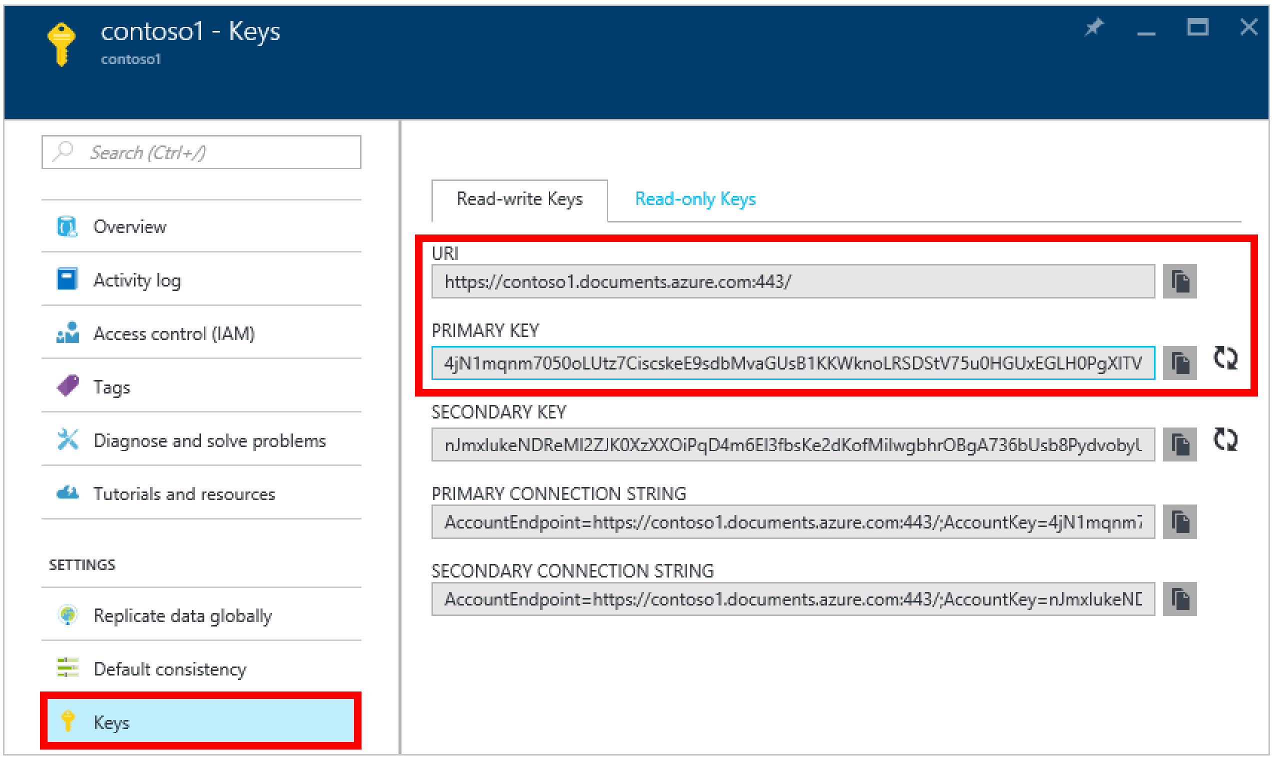
It provides write latencies, availability and read throughput comparable to that of eventual consistency but also provides the consistency guarantees that suit the needs of applications written to operate in the context of a user.

Click here, for more information on DocumentDB consistency levels.

View, copy, and regenerate access keys

When you create an Azure Cosmos DB account, the service generates two master access keys that can be used for authentication when the Azure Cosmos DB account is accessed. By providing two access keys, Azure Cosmos DB enables you to regenerate the keys with no interruption to your Azure Cosmos DB account.

In the Azure portal, access the **Keys** blade from the resource menu on the **Azure Cosmos DB account** blade to view, copy, and regenerate the access keys that are used to access your Azure Cosmos DB account.



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

PRIMARY KEY
4jN1mqnm7050oLUtz7CiscskeE9sdbMvaGUsB1KKWknoLRSDStV75u0HGUxEGLH0PgXITV

SECONDARY KEY
nJmxlukeNDReMI2ZJK0XzXXOiPqD4m6El3fbsKe2dKofMilwgbhrOBgA736bUsb8Pydvobyl

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

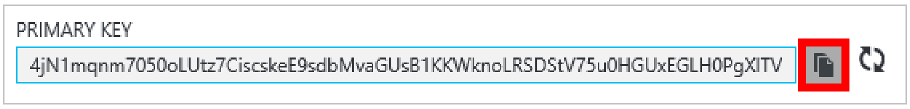
SECONDARY CONNECTION STRING
AccountEndpoint=https://contoso1.documents.azure.com:443/;AccountKey=nJmxlukeND

> 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.



PRIMARY KEY
4jN1mqnm7050oLUtz7CiscskeE9sdbMvaGUsB1KKWknoLRSDStV75u0HGUxEGLH0PgXITV

Regenerate access keys

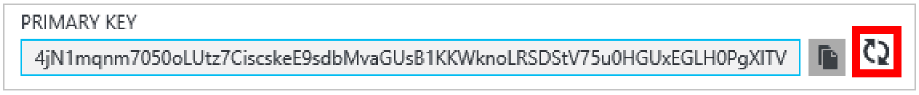
You should change the access keys to your Azure Cosmos DB account periodically to help keep your connections more secure. Two access keys are assigned to enable you to maintain connections to the Azure Cosmos DB account using one access key while you regenerate the other access key.

> 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 Azure Cosmos DB account must be updated to use the new key.

If you have applications or cloud services using the Azure Cosmos DB 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 Azure Cosmos DB account.
2. Regenerate the primary access key for your Azure Cosmos DB account. In the Azure Portal, access your Azure Cosmos DB account.
3. In the **Azure Cosmos DB Account** blade, click **Keys**.

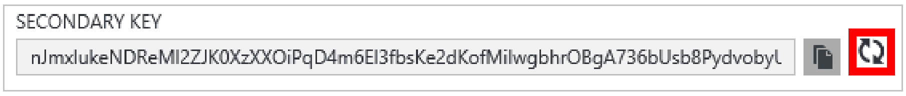
4. On the **Keys** blade, click the regenerate button, then click **Ok** to confirm that you want to generate a new key.



PRIMARY KEY

4jN1mqnm7050oLUtz7CiscskeE9sdbMvaGUsB1KKWknoLRSDStV75u0HGUxEGLH0PgXITV

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.



SECONDARY KEY

nJmxlukeNDReMI2ZJK0XzXXOiPqD4m6EI3fbsKe2dKofMilwgbhrOBgA736bUsb8Pydvobyl

NOTE

It can take several minutes before a newly generated key can be used to access your Azure Cosmos DB account.

Get the connection string

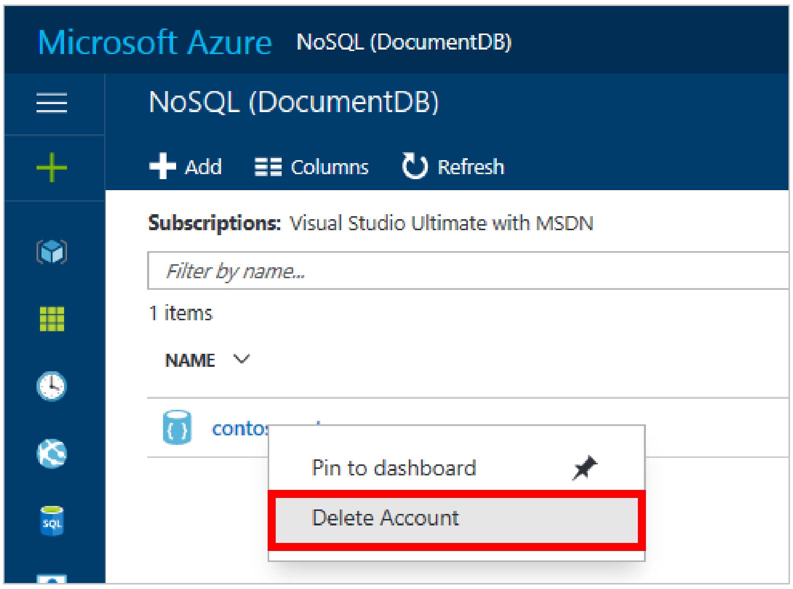
To retrieve your connection string, do the following:

1. In the Azure portal, access your Azure Cosmos DB 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 Azure Cosmos DB Database Migration Tool, append the database name to the end of the connection string. `AccountEndpoint=< >;AccountKey=< >;Database=< >`.

Delete an Azure Cosmos DB account

To remove an Azure Cosmos DB account from the Azure Portal that you are no longer using, right-click the account name, and click **Delete account**.

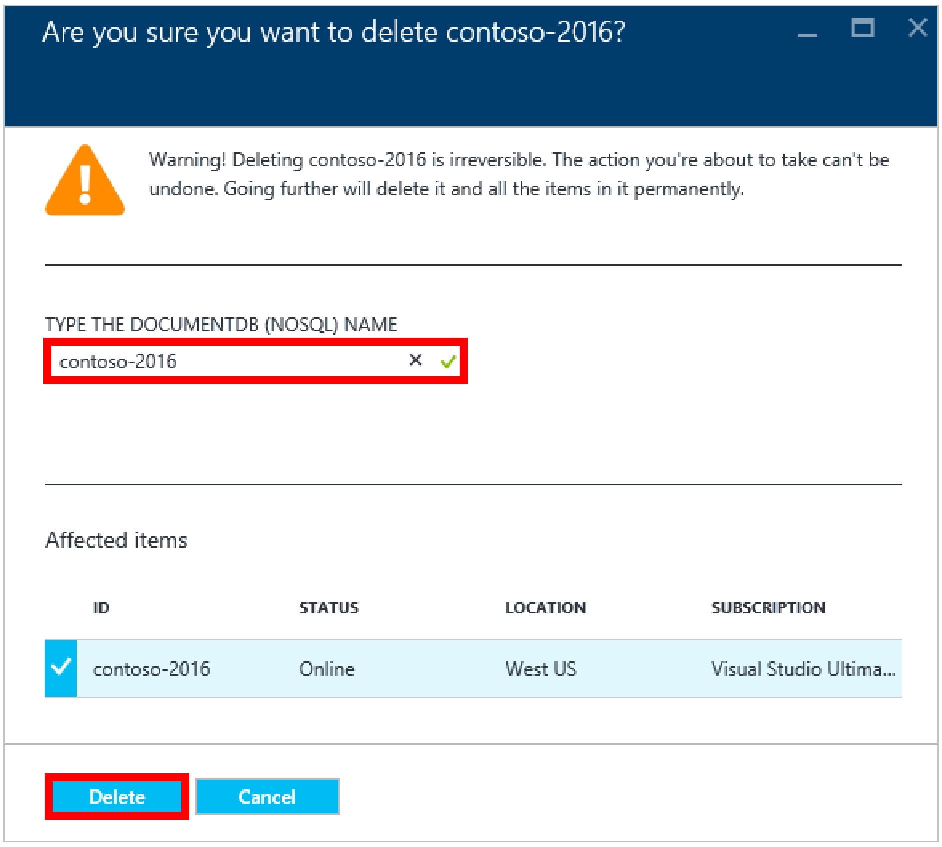


Subscriptions: Visual Studio Ultimate with MSDN

1 items

contos

1. In the Azure portal, access the Azure Cosmos DB account you wish to delete.
2. On the **Azure Cosmos DB account** blade, right-click the account, and then click **Delete Account**.
3. On the resulting confirmation blade, type the Azure Cosmos DB 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...

Next steps

Learn how to get started with your Azure Cosmos DB account.

Azure Cosmos DB database encryption at rest

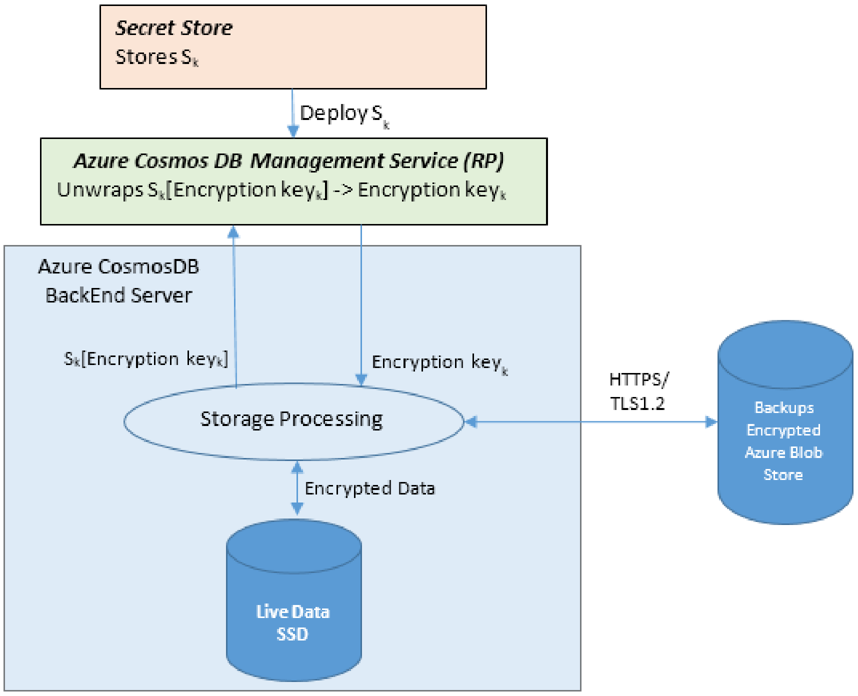
5/30/2017 • 3 min to read • Edit Online

Encryption at rest is a phrase that commonly refers to the encryption of data on nonvolatile storage devices, such as solid state drives (SSDs) and hard disk drives (HDDs). Cosmos DB stores its primary databases on SSDs. Its media attachments and backups are stored in Azure Blob storage, which is generally backed up by HDDs. With the release of encryption at rest for Cosmos DB, all your databases, media attachments, and backups are encrypted. Your data is now encrypted in transit (over the network) and at rest (nonvolatile storage), giving you end-to-end encryption.

As a PaaS service, Cosmos DB is very easy to use. Because all user data stored in Cosmos DB is encrypted at rest and in transport, you don't have to take any action. Another way to put this is that encryption at rest is "on" by default. There are no controls to turn it off or on. We provide this feature while we continue to meet our availability and performance SLAs.

Implement encryption at rest

Encryption at rest is implemented by using a number of security technologies, including secure key storage systems, encrypted networks, and cryptographic APIs. Systems that decrypt and process data have to communicate with systems that manage keys. The diagram shows how storage of encrypted data and the management of keys is separated.



Secret Store
Stores S_k
Deploy S_k
Azure Cosmos DB Management Service (RP)
Unwraps S_k[Encryption key_k] -> Encryption key_k
Azure CosmosDB BackEnd Server
S_k[Encryption key_k]
Encryption key_k
Storage Processing
Encrypted Data
Live Data SSD
HTTPS/ TLS1.2
Backups Encrypted Azure Blob Store

The basic flow of a user request is as follows:

- The user database account is made ready, and storage keys are retrieved via a request to the Management Service Resource Provider.
- A user creates a connection to Cosmos DB via HTTPS/secure transport. (The SDKs abstract the details.)
- The user sends a JSON document to be stored over the previously created secure connection.
- The JSON document is indexed unless the user has turned off indexing.
- Both the JSON document and index data are written to secure storage.

- Periodically, data is read from the secure storage and backed up to the Azure Encrypted Blob Store.

Frequently asked questions

Q: How much more does Azure Storage cost if Storage Service Encryption is enabled?

A: There is no additional cost.

Q: Who manages the encryption keys?

A: The keys are managed by Microsoft.

Q: How often are encryption keys rotated?

A: Microsoft has a set of internal guidelines for encryption key rotation, which Cosmos DB follows. The specific guidelines are not published. Microsoft does publish the Security Development Lifecycle (SDL), which is seen as a subset of internal guidance and has useful best practices for developers.

Q: Can I use my own encryption keys?

A: Cosmos DB is a PaaS service, and we worked hard to keep the service easy to use. We've noticed this question is often asked as a proxy question for meeting a compliance requirement like PCI-DSS. As part of building this feature, we worked with compliance auditors to ensure that customers who use Cosmos DB meet their requirements without the need to manage keys themselves. As a result, we currently do not offer users the option to burden themselves with key management.

Q: What regions have encryption turned on?

A: All Azure Cosmos DB regions have encryption turned on for all user data.

Q: Does encryption affect the performance latency and throughput SLAs?

A: There is no impact or changes to the performance SLAs now that encryption at rest is enabled for all existing and new accounts. You can read more on the SLA for Cosmos DB page to see the latest guarantees.

Q: Does the local emulator support encryption at rest?

A: The emulator is a standalone dev/test tool and does not use the key management services that the managed Cosmos DB service uses. Our recommendation is to enable BitLocker on drives where you are storing sensitive emulator test data. The emulator supports changing the default data directory as well as using a well-known location.

Next steps

For an overview of Cosmos DB security and the latest improvements, see Azure Cosmos DB database security. For more information about Microsoft certifications, see the Azure Trust Center.

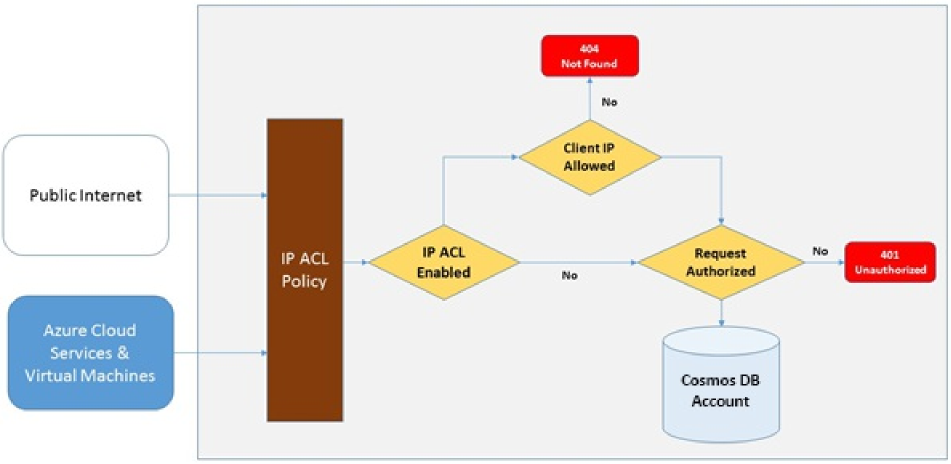
Azure Cosmos DB firewall support

5/30/2017 • 4 min to read • Edit Online

To secure data stored in an Azure Cosmos DB database account, Azure Cosmos DB 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, Azure Cosmos DB 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 Azure Cosmos DB database account. With this model, you can now configure an Azure Cosmos DB database account to be accessible only from an approved set of machines and/or cloud services. Access to Azure Cosmos DB resources from these approved sets of machines and services still require the caller to present a valid authorization token.

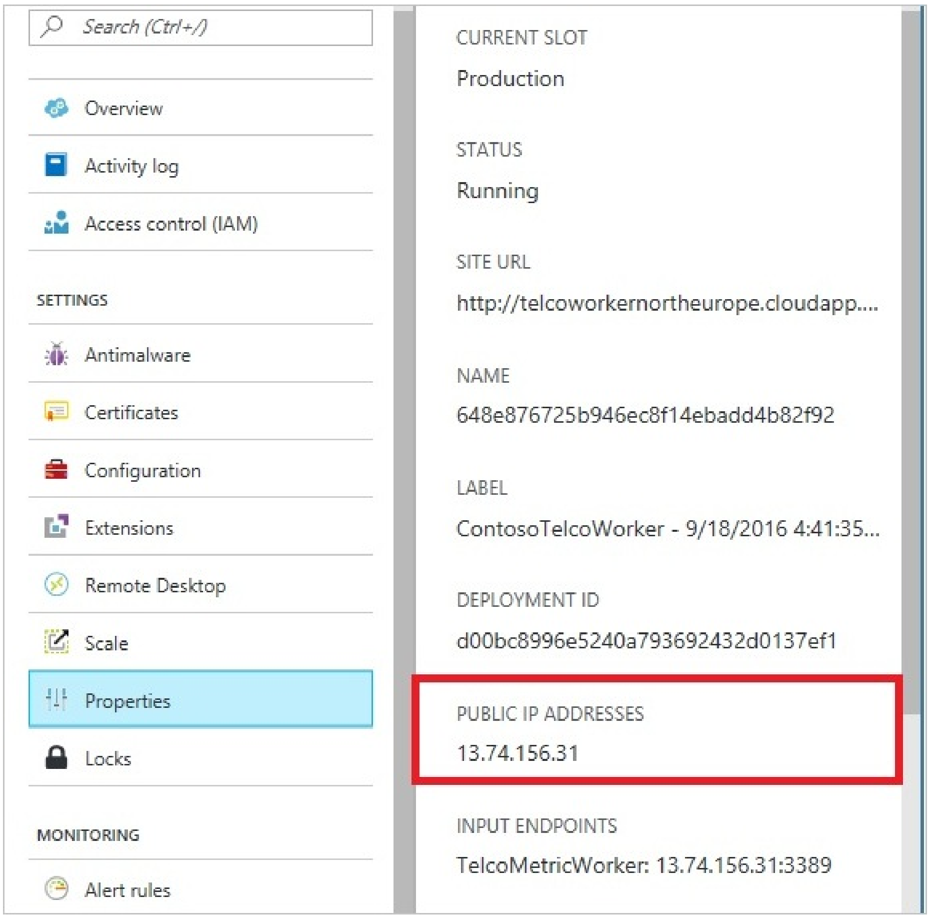
IP access control overview

By default, an Azure Cosmos DB 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 Azure Cosmos DB. To enable access to an Azure Cosmos DB 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 Azure Cosmos DB database account by configuring the IP access control policy. This ensures that all role instances of cloud services have access to your Azure Cosmos DB database account. You can retrieve IP addresses for your cloud services in the Azure portal, as shown in the following screenshot.



CURRENT SLOT
Production

STATUS
Running

SITE URL
http://telcoworkernortheurope.cloudapp....

NAME
648e876725b946ec8f14ebadd4b82f92

LABEL
ContosoTelcoWorker - 9/18/2016 4:41:35...

DEPLOYMENT ID
d00bc8996e5240a793692432d0137ef1

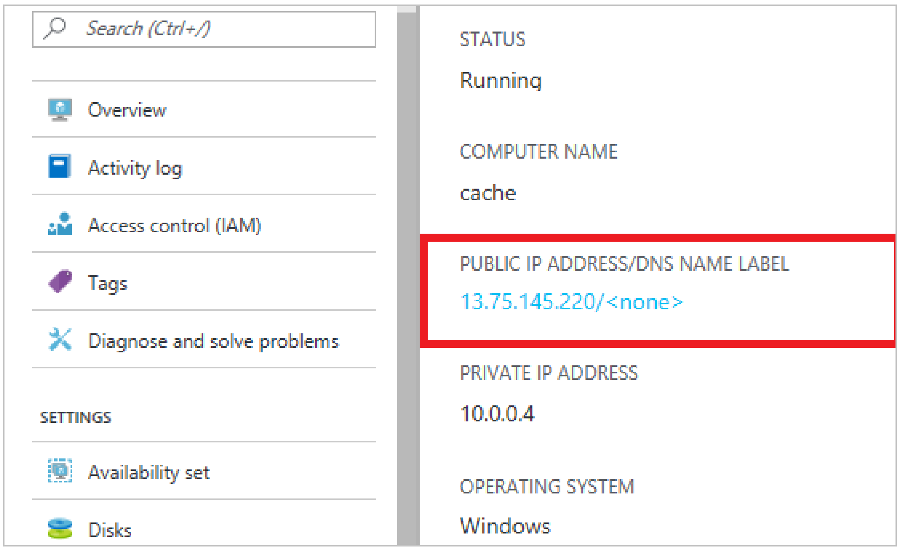
PUBLIC IP ADDRESSES
13.74.156.31

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 Azure Cosmos DB 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 Azure Cosmos DB. To configure the Azure Cosmos DB 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 Azure Cosmos DB database account by configuring the IP access control policy. You can retrieve IP addresses for virtual machines in the Azure portal, as shown in the following screenshot.



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 Azure Cosmos DB database account.

Connections from the internet

When you access an Azure Cosmos DB 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 Azure Cosmos DB database account.

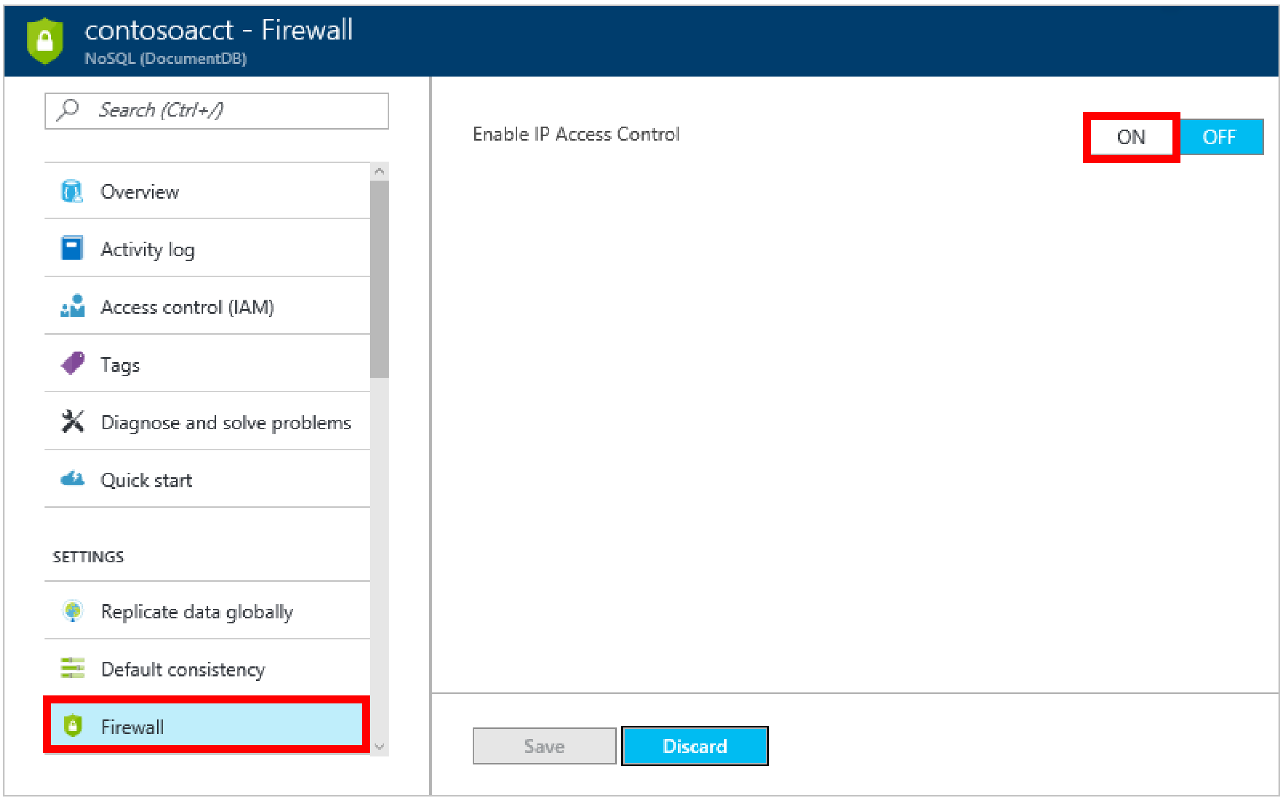
Configuring the IP access control policy

The IP access control policy can be set in the Azure portal, or programmatically through Azure CLI, Azure Powershell, or the REST API by updating the `ipRangeFilter` property. IP addresses/ranges must be comma separated and must not contain any spaces. Example: "13.91.6.132,13.91.6.1/24". When updating your database account through these methods, be sure to populate all of the properties to prevent resetting to default settings.

> NOTE
>
> By enabling an IP access control policy for your Azure Cosmos DB database account, all access to your Azure Cosmos DB 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.

To simplify development, the Azure portal helps you identify and add the IP of your client machine to the allowed list, so that apps running your machine can access the Azure Cosmos DB account. Note that the client IP address here is detected as seen by the portal. It may be the client IP address of your machine, but it could also be the IP address of your network gateway. Do not forget to remove it before going to production.

To set the IP access control policy in the Azure portal, navigate to the Azure Cosmos DB account blade, click **Firewall** in the navigation menu, then click **ON**

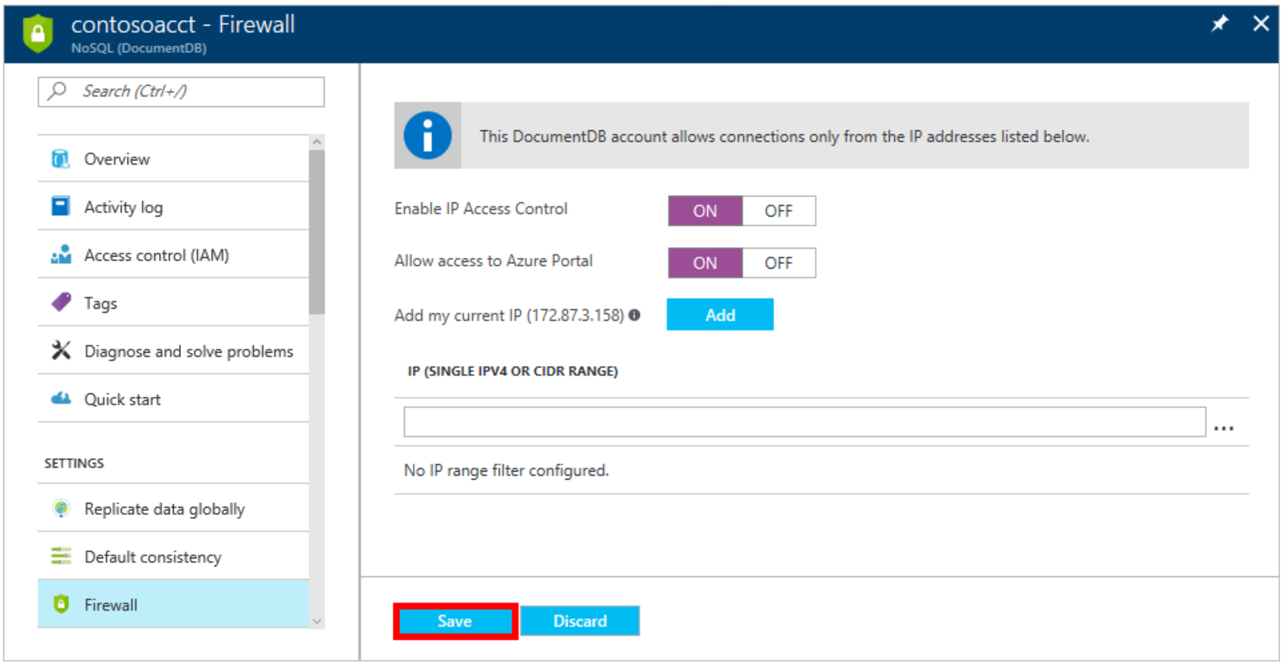


In the new pane, specify whether the Azure portal can access the account, and add other addresses and ranges as appropriate, then click **Save**.

NOTE

When you enable an IP access control policy, you need to add the IP address for the Azure portal to maintain access. The portal IP addresses are:

REGION	IP ADDRESS
All regions except those specified below	104.42.195.92
Germany	51.4.229.218
China	139.217.8.252
US Gov Arizona	52.244.48.71



This DocumentDB account allows connections only from the IP addresses listed below.

Enable IP Access Control: ON | OFF

Allow access to Azure Portal: ON | OFF

Add my current IP (172.87.3.158)

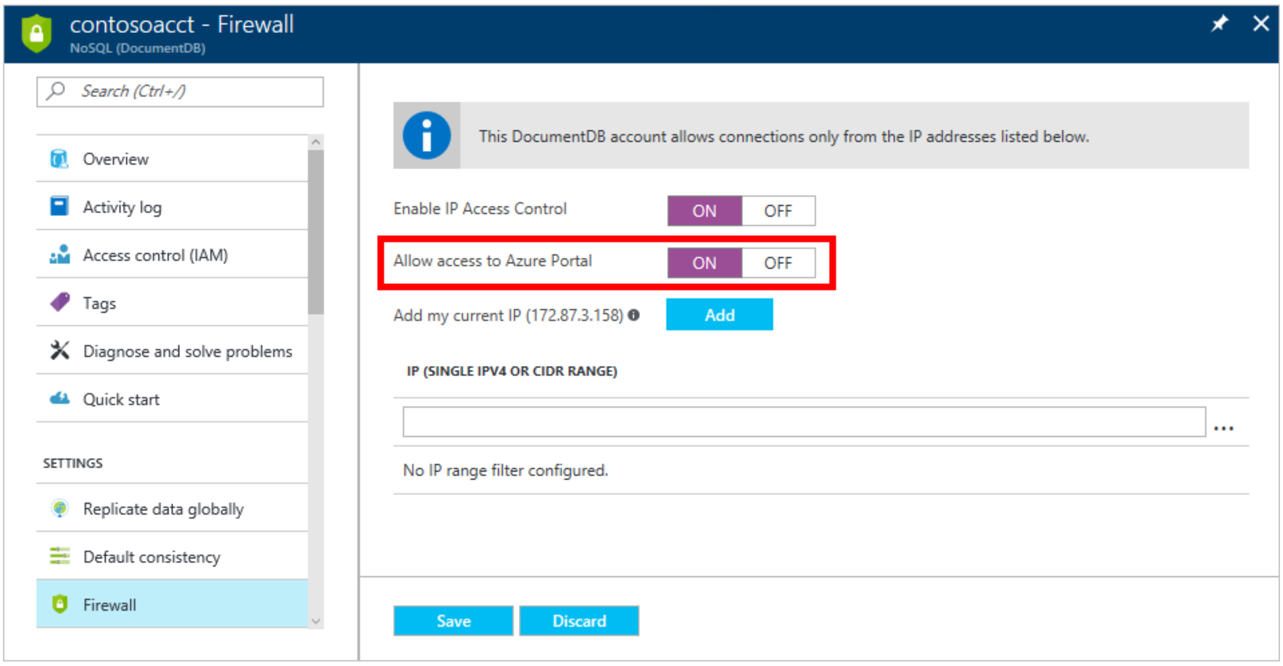
IP (SINGLE IPV4 OR CIDR RANGE)

No IP range filter configured.

Troubleshooting the IP access control policy

Portal operations

By enabling an IP access control policy for your Azure Cosmos DB database account, all access to your Azure Cosmos DB database account from machines outside the configured allowed list of IP address ranges are blocked. Therefore if you want to enable portal data plane operations like browsing collections and query documents, you need to explicitly allow Azure portal access using the **Firewall** blade in the portal.



contosoacct - Firewall

NoSQL (DocumentDB)

This DocumentDB account allows connections only from the IP addresses listed below.

Enable IP Access Control: ON | OFF

Allow access to Azure Portal: ON | OFF

Add my current IP (172.87.3.158) Add

IP (SINGLE IPV4 OR CIDR RANGE)

No IP range filter configured.

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 Azure Cosmos DB database account to ensure the correct policy configuration is applied to your Azure Cosmos DB database account.

Next steps

For information about network related performance tips, see Performance tips.

Securing access to Azure Cosmos DB data

6/6/2017 • 6 min to read • Edit Online

This article provides an overview of securing access to data stored in Microsoft Azure Cosmos DB.

Azure Cosmos DB uses two types of keys to authenticate users and provide access to its data and resources.

KEY TYPE	RESOURCES
Master keys	Used for administrative resources: database accounts, databases, users, and permissions
Resource tokens	Used for application resources: collections, documents, attachments, stored procedures, triggers, and UDFs

Master keys

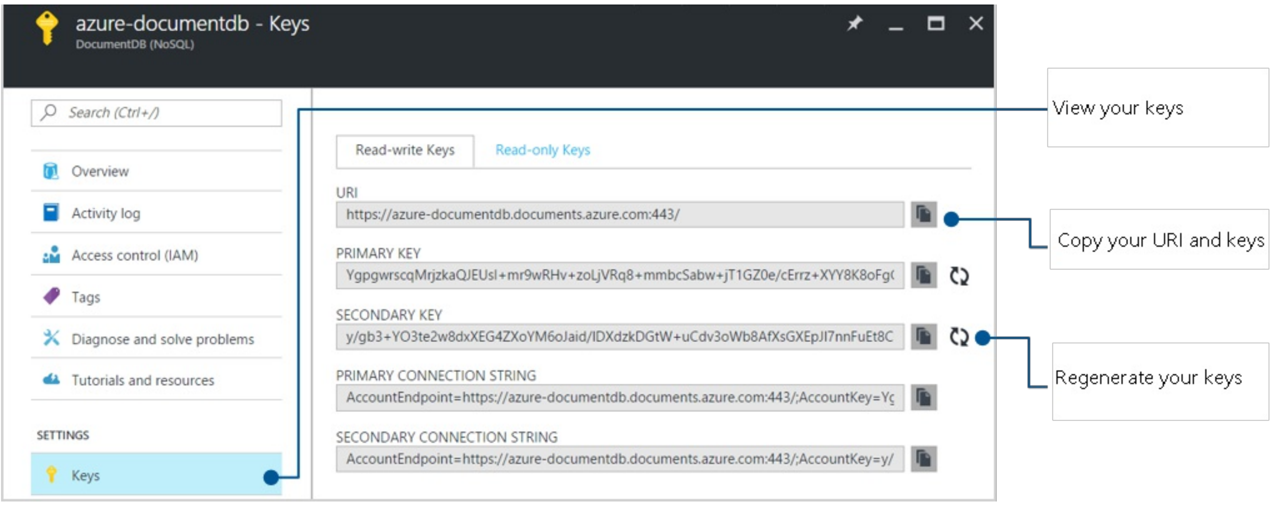
Master keys provide access to the all the administrative resources for the database account. Master keys:

- Provide access to accounts, databases, users, and permissions.
- Cannot be used to provide granular access to collections and documents.
- Are created during the creation of an account.
- Can be regenerated at any time.

Each account consists of two Master keys: a primary key and secondary key. The purpose of dual keys is so that you can regenerate, or roll keys, providing continuous access to your account and data.

In addition to the two master keys for the Cosmos DB account, there are two read-only keys. These read-only keys only allow read operations on the account. Read-only keys do not provide access to read permissions resources.

Primary, secondary, read only, and read-write master keys can be retrieved and regenerated using the Azure portal. For instructions, see View, copy, and regenerate access keys.



URI
https://azure-documentdb.documents.azure.com:443/

PRIMARY KEY
YgpgwrscqMrjzkaQJEUsl+mr9wRHv+zoLjVRq8+mmbcSabw+jT1GZ0e/cErrz+XYY8K8oFg(

SECONDARY KEY
y/gb3+YO3te2w8dxXEG4ZXoYM6oJaid/IDXdzkDGtW+uCdv3oWb8AfXsGXEpJl7nnFuEt8C

PRIMARY CONNECTION STRING
AccountEndpoint=https://azure-documentdb.documents.azure.com:443/;AccountKey=Yç

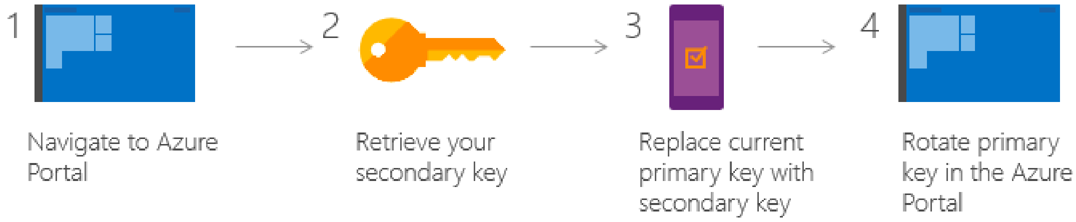
SECONDARY CONNECTION STRING
AccountEndpoint=https://azure-documentdb.documents.azure.com:443/;AccountKey=y/

View your keys

Copy your URI and keys

Regenerate your keys

The process of rotating your master key is simple. Navigate to the Azure portal to retrieve your secondary key, then replace your primary key with your secondary key in your application, then rotate the primary key in the Azure portal.


1
Navigate to Azure Portal
2
Retrieve your secondary key
3
Replace current primary key with secondary key
4
Rotate primary key in the Azure Portal

Code sample to use a master key

The following code sample illustrates how to use a Cosmos DB account endpoint and master key to instantiate a DocumentClient and create a database.

```
//Read the DocumentDB endpointUrl and authorization keys from config.
//These values are available from the Azure portal on the Azure Cosmos DB 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
    });
```

Resource tokens

Resource tokens provide access to the application resources within a database. Resource tokens:

- Provide access to specific collections, partition keys, documents, attachments, stored procedures, triggers, and UDFs.
- Are created when a user is granted permissions to a specific resource.
- Are recreated when a permission resource is acted upon on by POST, GET, or PUT call.
- Use a hash resource token specifically constructed for the user, resource, and permission.
- Are time bound with a customizable validity period. The default valid timespan is one hour. Token lifetime, however, may be explicitly specified, up to a maximum of five hours.
- Provide a safe alternative to giving out the master key.
- Enable clients to read, write, and delete resources in the Cosmos DB account according to the permissions they've been granted.

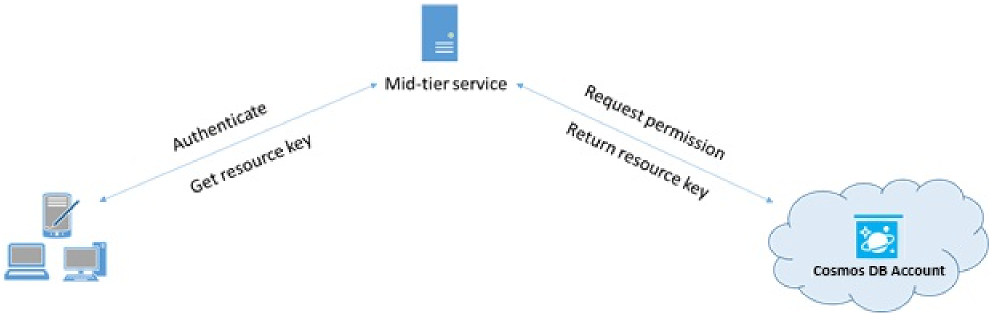
You can use a resource token (by creating Cosmos DB users and permissions) when you want to provide access to resources in your Cosmos DB account to a client that cannot be trusted with the master key.

Cosmos DB resource tokens provide a safe alternative that enables clients to read, write, and delete resources in your Cosmos DB 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 Cosmos DB 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 Cosmos DB 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 receive a 401 unauthorized exception. At this point, the phone app re-establishes the identity and requests a new resource token.



Resource token generation and management is handled by the native Cosmos DB client libraries; however, if you use REST you must construct the request/authentication headers. For more information on creating authentication headers for REST, see Access Control on Cosmos DB Resources or the source code for our SDKs.

For an example of a middle tier service used to generate or broker resource tokens, see the ResourceTokenBroker app.

Users

Cosmos DB users are associated with a Cosmos DB database. Each database can contain zero or more Cosmos DB users. The following code sample shows how to create a Cosmos DB user resource.

```
//Create a user.
User docUser = new User
{
   Id = "mobileuser"
};

docUser = await client.CreateUserAsync(UriFactory.CreateDatabaseUri("db"), docUser);
```

NOTE

Each Cosmos DB user has a PermissionsLink property that can be used to retrieve the list of permissions associated with the user.

Permissions

A Cosmos DB permission resource is associated with a Cosmos DB user. Each user may contain zero or more Cosmos DB 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 that 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 Cosmos DB stored procedures the user must have the All permission on the collection in which the stored procedure will be run.

Code sample to create permission

The following code sample 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.

Code sample to read permissions for user

To easily obtain all permission resources associated with a particular user, Cosmos DB 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);
```

Next steps

- To learn more about Cosmos DB database security, see Cosmos DB: Database security.
- To learn about managing master and read-only keys, see How to manage an Azure Cosmos DB account.
- To learn how to construct Azure Cosmos DB authorization tokens, see Access Control on Azure Cosmos DB Resources.

Accelerate real-time big-data analytics with the Spark to Azure Cosmos DB connector

6/14/2017 • 10 min to read • Edit Online

The Spark to Azure Cosmos DB connector enables Cosmos DB to act as an input source or output sink for Apache Spark jobs. Connecting Spark to Azure Cosmos DB accelerates your ability to solve fast-moving data science problems where you can use Cosmos DB to quickly persist and query data. The Spark to Azure Cosmos DB connector efficiently utilizes the native Cosmos DB managed indexes. The indexes enable updateable columns when you perform analytics and push-down predicate filtering against fast-changing globally distributed data, which range from Internet of Things (IoT) to data science and analytics scenarios.

For working with Spark GraphX and the Gremlin graph APIs of Azure Cosmos DB, see Perform graph analytics using Spark and Apache TinkerPop Gremlin.

Download

To get started, download the Spark to Azure Cosmos DB connector (preview) from the azure-cosmosdb-spark repository on GitHub.

Connector components

The connector utilizes the following components:

- Azure Cosmos DB enables customers to provision and elastically scale both throughput and storage across any number of geographical regions. The service offers:
  - Turn key global distribution and horizontal scale
  - Guaranteed single digit latencies at the 99th percentile
  - Multiple well-defined consistency models
  - Guaranteed high availability with multi-homing capabilities

  All features are backed by industry-leading, comprehensive service level agreements (SLAs).
- Apache Spark is a powerful open source processing engine that's built around speed, ease of use, and sophisticated analytics.
- Apache Spark on Azure HDInsight so that you can deploy Apache Spark in the cloud for mission-critical deployments by using Azure HDInsight.

Officially supported versions:

COMPONENT	VERSION
Apache Spark	2.0+
Scala	2.11
Azure DocumentDB Java SDK	1.10.0

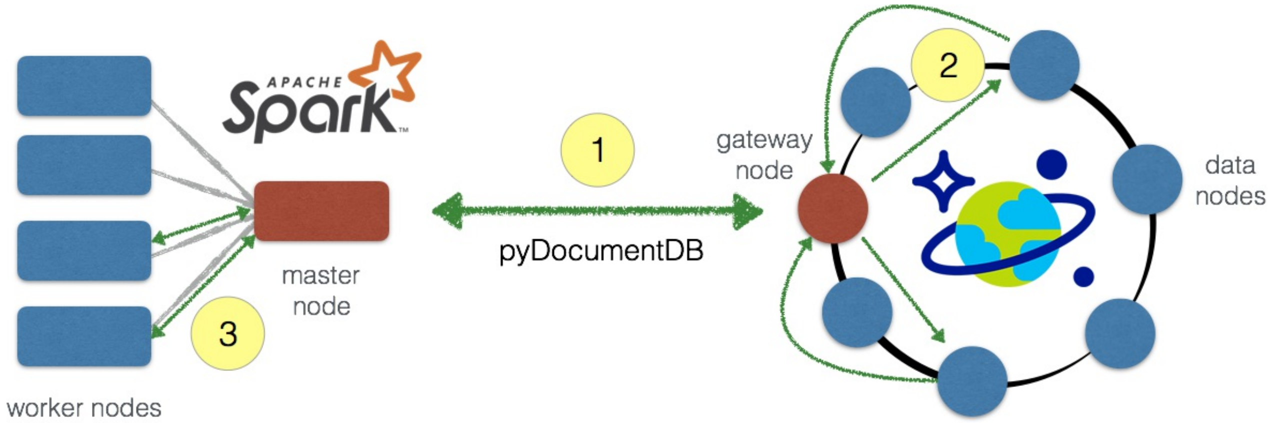
This article helps you run some simple samples by using Python (via pyDocumentDB) and the Scala interfaces.

There are two approaches to connect Apache Spark and Azure Cosmos DB:

- Use pyDocumentDB via the Azure DocumentDB Python SDK.
- Create a Java-based Spark to Cosmos DB connector by utilizing the Azure DocumentDB Java SDK.

pyDocumentDB implementation

The current pyDocumentDB SDK enables you to connect Spark to Cosmos DB as shown in the following diagram:



Data flow of the pyDocumentDB implementation

The data flow is as follows:

1. The Spark master node connects to the Cosmos DB gateway node via pyDocumentDB. A user specifies only the Spark and Cosmos DB connections. Connections to the respective master and gateway nodes are transparent to the user.
2. The gateway node makes the query against Cosmos DB where the query subsequently runs against the collection's partitions in the data nodes. The response for those queries is sent back to the gateway node, and that result set is returned to the Spark master node.
3. Subsequent queries (for example, against a Spark DataFrame) are sent to the Spark worker nodes for processing.

Communication between Spark and Cosmos DB is limited to the Spark master node and Cosmos DB gateway nodes. The queries go as fast as the transport layer between these two nodes allows.

Install pyDocumentDB

You can install pyDocumentDB on your driver node by using **pip**, for example:

```
pip install pyDocumentDB
```

Connect Spark to Cosmos DB via pyDocumentDB

The simplicity of the communication transport makes execution of a query from Spark to Cosmos DB by using pyDocumentDB relatively simple.

The following code snippet shows how to use pyDocumentDB in a Spark context.

```
# Import Necessary Libraries
import pydocumentdb
from pydocumentdb import document_client
from pydocumentdb import documents
import datetime

# Configuring the connection policy (allowing for endpoint discovery)
connectionPolicy = documents.ConnectionPolicy()
connectionPolicy.EnableEndpointDiscovery
connectionPolicy.PreferredLocations = ["Central US", "East US 2", "Southeast Asia", "Western Europe","Canada Central"]


# Set keys to connect to Cosmos DB
masterKey = 'le1n99i1w5l7uvokJs3RT5ZAH8dc3ql7lx2CG0h0kK4lVWPkQnwpRLyAN0nwS1z4Cyd1lJgvGUfMWR3v8vkXKA=='
host = 'https://doctorwho.documents.azure.com:443/'
client = document_client.DocumentClient(host, {'masterKey': masterKey}, connectionPolicy)
```

As noted in the code snippet:

- The Cosmos DB Python SDK (`pyDocumentDB`) contains the all the necessary connection parameters. For example, the preferred locations parameter chooses the read replica and priority order.
- Import the necessary libraries and configure your **masterKey** and **host** to create the Cosmos DB *client* (**pydocumentdb.document_client**).

Execute Spark Queries via pyDocumentDB

The following examples use the Cosmos DB instance that was created in the previous snippet by using the specified read-only keys. The following code snippet connects to the **airports.codes** collection in the DoctorWho account as specified earlier and runs a query to extract the airport cities in Washington state.

```
# Configure Database and Collections
databaseId = 'airports'
collectionId = 'codes'

# Configurations the Cosmos DB client will use to connect to the database and collection
dbLink = 'dbs/' + databaseId
collLink = dbLink + '/colls/' + collectionId

# Set query parameter
querystr = "SELECT c.City FROM c WHERE c.State='WA'"

# Query documents
query = client.QueryDocuments(collLink, querystr, options=None, partition_key=None)

# Query for partitioned collections
# query = client.QueryDocuments(collLink, query, options= { 'enableCrossPartitionQuery': True }, partition_key=None)

# Push into list `elements`
elements = list(query)
```

After the query has been executed via **query**, the result is a **query_iterable.QueryIterable** that is converted to a Python list. A Python list can be easily converted to a Spark DataFrame by using the following code:

```
# Create `df` Spark DataFrame from `elements` Python list
df = spark.createDataFrame(elements)
```

Why use the pyDocumentDB to connect Spark to Cosmos DB?

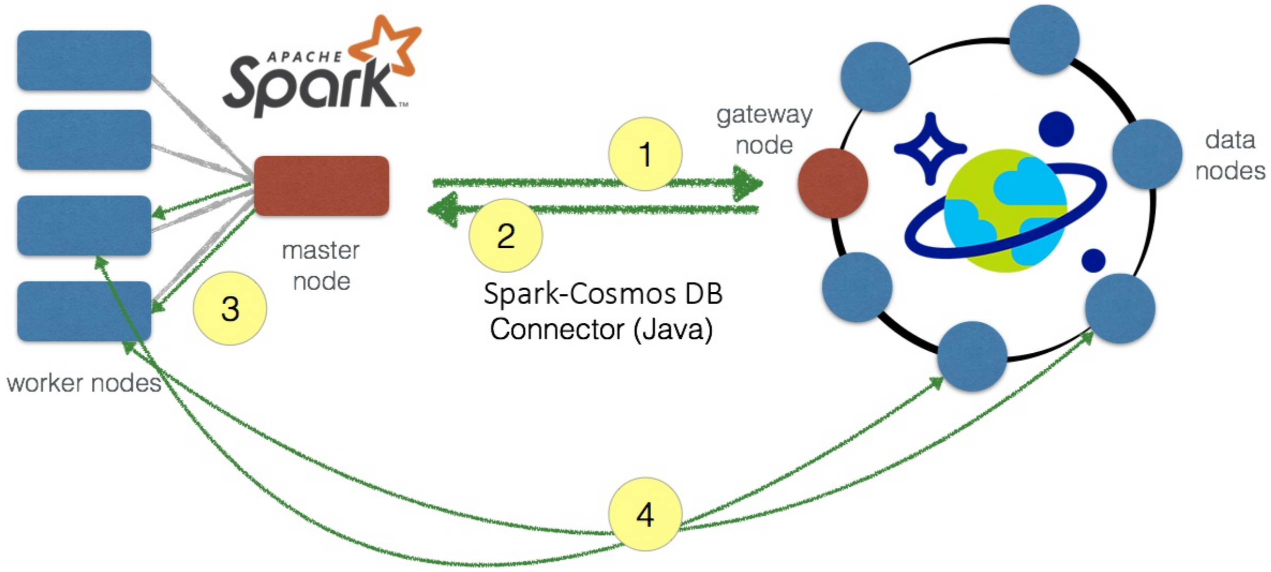
Connecting Spark to Cosmos DB by using pyDocumentDB is typically for scenarios where:

- You want to use Python.

- You are returning a relatively small result set from Cosmos DB to Spark. Note that the underlying dataset in Cosmos DB can be quite large. You are applying filters, that is, running predicate filters, against your Cosmos DB source.

Spark to Cosmos DB connector

The Spark to Cosmos DB connector utilizes the Azure DocumentDB Java SDK and moves data between the Spark worker nodes and Cosmos DB as shown in the following diagram:



The data flow is as follows:

1. The Spark master node connects to the Cosmos DB gateway node to obtain the partition map. A user specifies only the Spark and Cosmos DB connections. Connections to the respective master and gateway nodes are transparent to the user.
2. This information is provided back to the Spark master node. At this point, you should be able to parse the query to determine the partitions and their locations in Cosmos DB that you need to access.
3. This information is transmitted to the Spark worker nodes.
4. The Spark worker nodes connect to the Cosmos DB partitions directly to extract the data and return the data to the Spark partitions in the Spark worker nodes.

Communication between Spark and Cosmos DB is significantly faster because the data movement is between the Spark worker nodes and the Cosmos DB data nodes (partitions).

Build the Spark to Cosmos DB connector

Currently, the connector project uses maven. To build the connector without dependencies, you can run:

```
mvn clean package
```

You can also download the latest versions of the JAR from the *releases* folder.

Include the Azure Cosmos DB Spark JAR

Before you execute any code, you need to include the Azure Cosmos DB Spark JAR. If you are using the **spark-shell**, then you can include the JAR by using the **--jars** option.

```
spark-shell --master $master --jars /$location/azure-cosmosdb-spark-0.0.3-jar-with-dependencies.jar
```

If you want to execute the JAR without dependencies, use the following code:

```
spark-shell --master $master --jars /$location/azure-cosmosdb-spark-0.0.3.jar,/$location/azure-documentdb-1.10.0.jar
```

If you are using a notebook service such as Azure HDInsight Jupyter notebook service, you can use the **spark magic** commands:

```
%%configure
{ "jars": ["wasb:///example/jars/azure-documentdb-1.10.0.jar","wasb:///example/jars/azure-cosmosdb-spark-0.0.3.jar"],
  "conf": {
    "spark.jars.excludes": "org.scala-lang:scala-reflect"
   }
}
```

The **jars** command enables you to include the two JARs that are needed for **azure-cosmosdb-spark** (itself and the Azure DocumentDB Java SDK) and exclude **scala-reflect** so that it does not interfere with the Livy calls (Jupyter notebook > Livy > Spark).

Connect Spark to Cosmos DB using the connector

Although the communication transport is a little more complicated, executing a query from Spark to Cosmos DB by using the connector is significantly faster.

The following code snippet shows how to use the connector in a Spark context.

```
// Import Necessary Libraries
import org.joda.time._
import org.joda.time.format._
import com.microsoft.azure.cosmosdb.spark.schema._
import com.microsoft.azure.cosmosdb.spark._
import com.microsoft.azure.cosmosdb.spark.config.Config

// Configure connection to your collection
val readConfig2 = Config(Map("Endpoint" -> "https://doctorwho.documents.azure.com:443/",
"Masterkey" -> "le1n99i1w5l7uvokJs3RT5ZAH8dc3ql7lx2CG0h0kK4lVWPkQnwpRLyAN0nwS1z4Cyd1lJgvGUfMWR3v8vkXKA==",
"Database" -> "DepartureDelays",
"preferredRegions" -> "Central US;East US2;",
"Collection" -> "flights_pcoll",
"SamplingRatio" -> "1.0"))

// Create collection connection
val coll = spark.sqlContext.read.cosmosDB(readConfig2)
coll.createOrReplaceTempView("c")
```

As noted in the code snippet:

- **azure-cosmosdb-spark** contains the all the necessary connection parameters, which include the preferred locations. For example, you can choose the read replica and priority order.
- Just import the necessary libraries and configure your masterKey and host to create the Cosmos DB client.

Execute Spark queries via the connector

The following example uses the Cosmos DB instance that was created in the previous snippet by using the specified read-only keys. The following code snippet connects to the DepartureDelays.flights_pcoll collection (in the DoctorWho account as specified earlier) and runs a query to extract the flight delay information of flights that are departing from Seattle.

```
// Queries
var query = "SELECT c.date, c.delay, c.distance, c.origin, c.destination FROM c WHERE c.origin = 'SEA'"
val df = spark.sql(query)

// Run DF query (count)
df.count()

// Run DF query (show)
df.show()
```

Why use the Spark to Cosmos DB connector implementation?

Connecting Spark to Cosmos DB by using the connector is typically for scenarios where:

- You want to use Scala and update it to include a Python wrapper as noted in Issue 3: Add Python wrapper and examples.
- You have a large amount of data to transfer between Apache Spark and Cosmos DB.

To give you an idea of the query performance difference, see the Query Test Runs wiki.

Distributed aggregation example

This section provides some examples of how you can do distributed aggregations and analytics by using Apache Spark and Azure Cosmos DB together. Azure Cosmos DB already supports aggregations, which is discussed in the Planet scale aggregates with Azure Cosmos DB blog. Here is how you can take it to the next level with Apache Spark.

Note that these aggregations are in reference to the Spark to Cosmos DB Connector notebook.

Connect to flights sample data

These aggregation examples access some flight performance data that's stored in our **DoctorWho** Cosmos DB database. To connect to it, you need to utilize the following code snippet:

```
// Import Spark to Cosmos DB connector
import com.microsoft.azure.cosmosdb.spark.schema._
import com.microsoft.azure.cosmosdb.spark._
import com.microsoft.azure.cosmosdb.spark.config.Config

// Connect to Cosmos DB Database
val readConfig2 = Config(Map("Endpoint" -> "https://doctorwho.documents.azure.com:443/",
"Masterkey" -> "le1n99i1w5l7uvokJs3RT5ZAH8dc3ql7lx2CG0h0kK4lVWPkQnwpRLyAN0nwS1z4Cyd1lJgvGUfMWR3v8vkXKA==",
"Database" -> "DepartureDelays",
"preferredRegions" -> "Central US;East US 2;",
"Collection" -> "flights_pcoll",
"SamplingRatio" -> "1.0"))

// Create collection connection
val coll = spark.sqlContext.read.cosmosDB(readConfig2)
coll.createOrReplaceTempView("c")
```

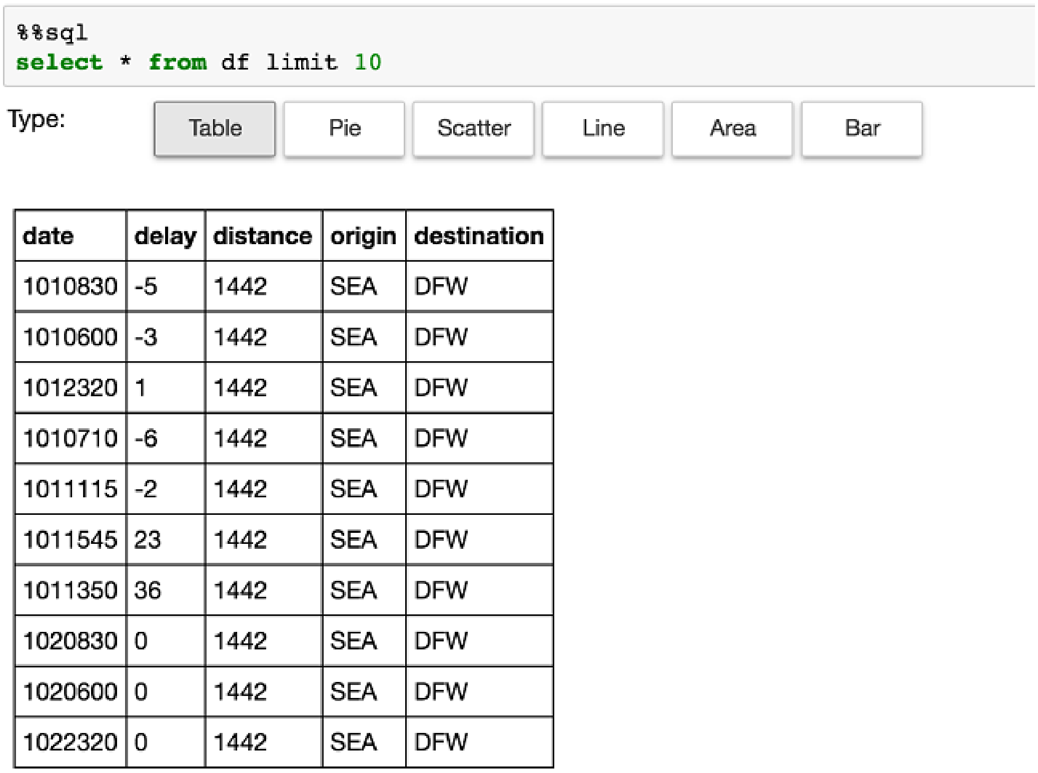
With this snippet, we are also going to run a base query that transfers the filtered set of data from Cosmos DB to Spark where the latter can perform distributed aggregates. In this case, we are asking for flights that depart from Seattle (SEA).

```
// Run, get row count, and time query
val originSEA = spark.sql("SELECT c.date, c.delay, c.distance, c.origin, c.destination FROM c WHERE c.origin = 'SEA'")
originSEA.createOrReplaceTempView("originSEA")
```

The following results were generated by running the queries from the Jupyter notebook service. Note that all the code snippets are generic and not specific to any service.

Running LIMIT and COUNT queries

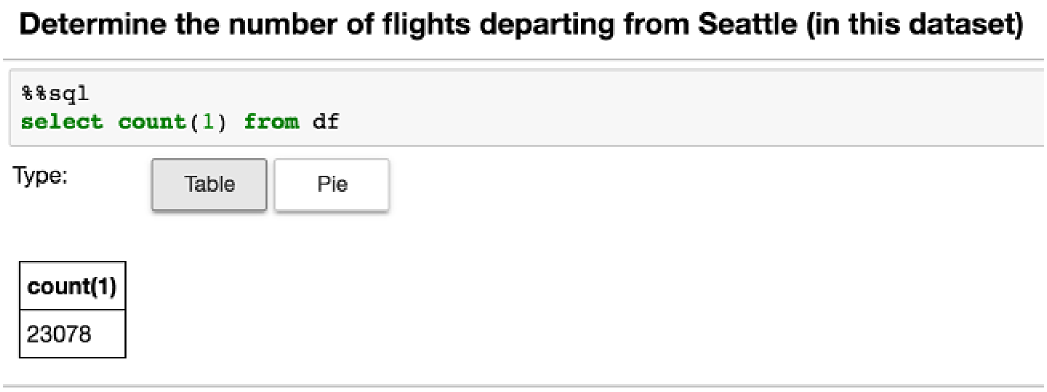
Just like you're used to in SQL/Spark SQL, let's start off with a **LIMIT** query:



```
%%sql
select * from df limit 10
```

date	delay	distance	origin	destination
1010830	-5	1442	SEA	DFW
1010600	-3	1442	SEA	DFW
1012320	1	1442	SEA	DFW
1010710	-6	1442	SEA	DFW
1011115	-2	1442	SEA	DFW
1011545	23	1442	SEA	DFW
1011350	36	1442	SEA	DFW
1020830	0	1442	SEA	DFW
1020600	0	1442	SEA	DFW
1022320	0	1442	SEA	DFW

The next query is a simple and fast **COUNT** query:



Determine the number of flights departing from Seattle (in this dataset)

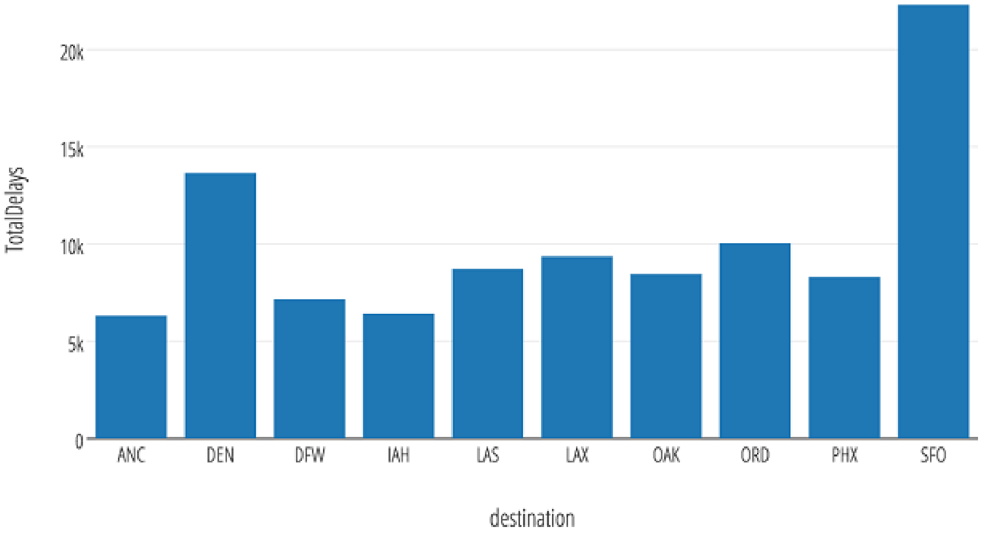
```
%%sql
select count(1) from df
```

count(1)
23078

GROUP BY query

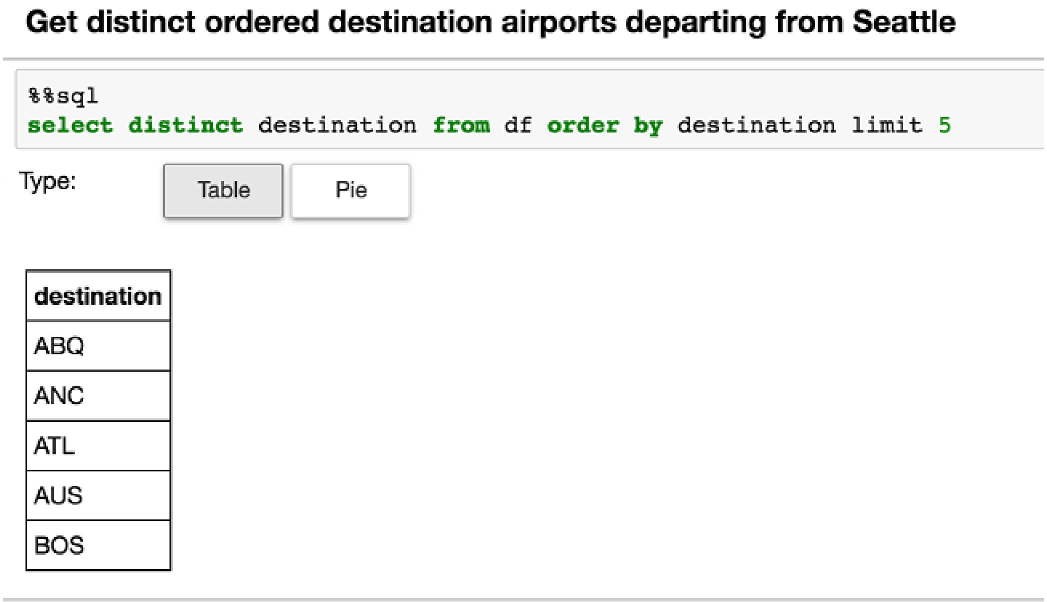
In this next set, we can easily run **GROUP BY** queries against our Cosmos DB database:

```
select destination, sum(delay) as TotalDelays
from originSEA
group by destination
order by sum(delay) desc limit 10
```



DISTINCT, ORDER BY query

And here is a **DISTINCT, ORDER BY** query:



Get distinct ordered destination airports departing from Seattle

```
%%sql
select distinct destination from df order by destination limit 5
```

Type: Table Pie

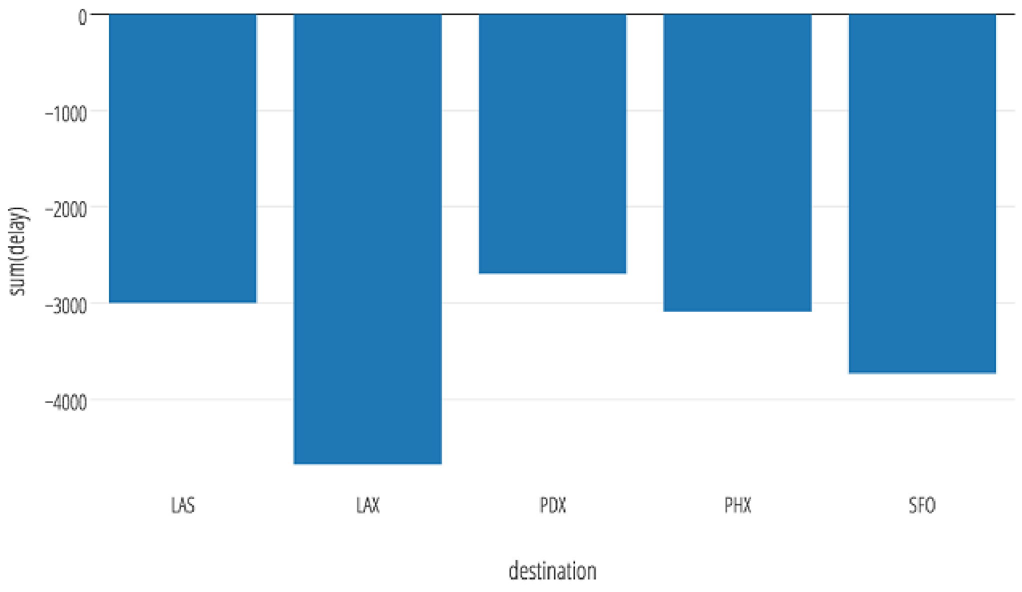
destination
ABQ
ANC
ATL
AUS
BOS

Continue the flight data analysis

You can use the following example queries to continue analysis of the flight data:

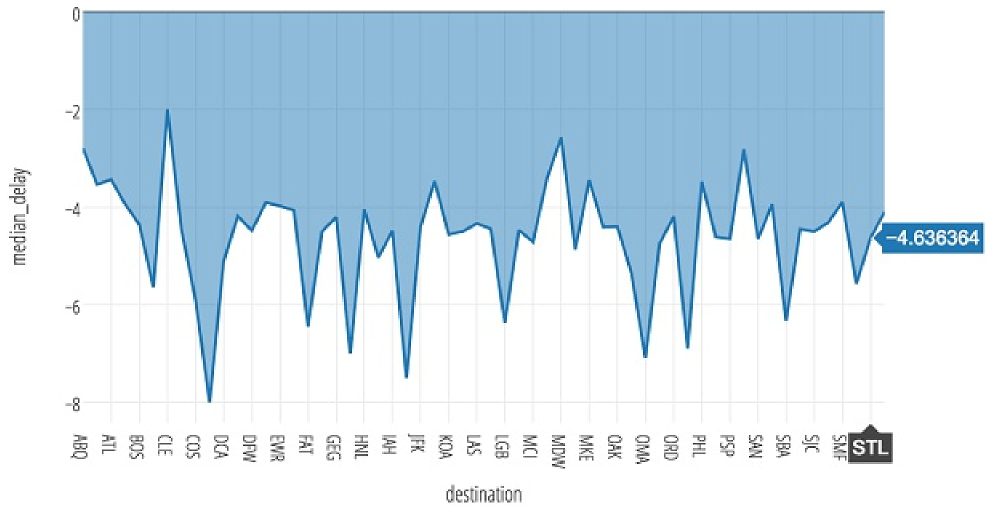
Top 5 delayed destinations (cities) departing from Seattle

```
select destination, sum(delay)
from originSEA
where delay < 0
group by destination
order by sum(delay) limit 5
```



Calculate median delays by destination cities departing from Seattle

```
select destination, percentile_approx(delay, 0.5) as median_delay
from originSEA
where delay < 0
group by destination
order by percentile_approx(delay, 0.5)
```



Next steps

If you haven't already, download the Spark to Cosmos DB connector from the azure-cosmosdb-spark GitHub repository and explore the additional resources in the repo:

- Distributed Aggregations Examples
- Sample Scripts and Notebooks

You might also want to review the Apache Spark SQL, DataFrames, and Datasets Guide and the Apache Spark on Azure HDInsight article.

Azure Cosmos DB: Perform graph analytics by using Spark and Apache TinkerPop Gremlin

6/12/2017 • 9 min to read • Edit Online

Azure Cosmos DB is the globally distributed, multi-model database service from Microsoft. You can create and query document, key/value, and graph databases, all of which benefit from the global-distribution and horizontal-scale capabilities at the core of Azure Cosmos DB. Azure Cosmos DB supports online transaction processing (OLTP) graph workloads that use Apache TinkerPop Gremlin.

Spark is an Apache Software Foundation project that's focused on general-purpose online analytical processing (OLAP) data processing. Spark provides a hybrid in-memory/disk-based distributed computing model that is similar to the Hadoop MapReduce model. You can deploy Apache Spark in the cloud by using Azure HDInsight.

By combining Azure Cosmos DB and Spark, you can perform both OLTP and OLAP workloads when you use Gremlin. This quick-start article demonstrates how to run Gremlin queries against Azure Cosmos DB on an Azure HDInsight Spark cluster.

Prerequisites

Before you can run this sample, you must have the following prerequisites:

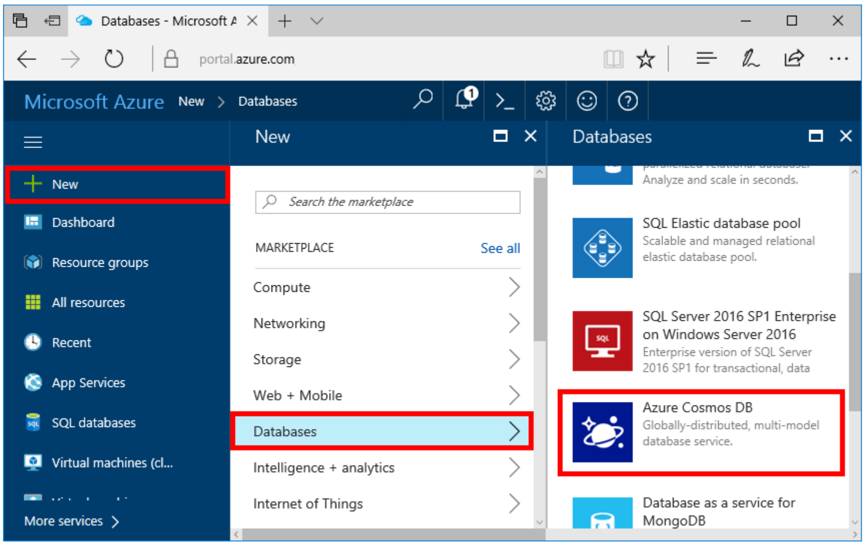
- Azure HDInsight Spark cluster 2.0
- JDK 1.8+ (If you don't have JDK, run `apt-get install default-jdk`.)
- Maven (If you don't have Maven, run `apt-get install maven`.)
- An Azure subscription (If you don't have an Azure subscription, create a free account before you begin.)

For information about how to set up an Azure HDInsight Spark cluster, see Provisioning HDInsight clusters.

Create an Azure Cosmos DB database account

First, create a database account with the Graph API by doing the following:

1. In a new window, sign in to the Azure portal.
2. In the left pane, click **New**, click **Databases**, and then click **Azure Cosmos DB**.

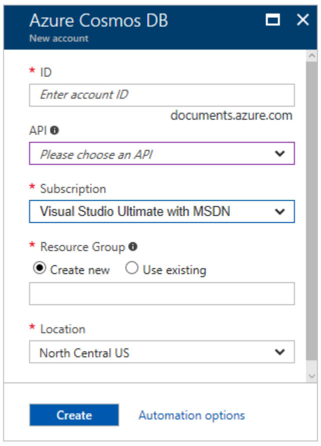


3. On the **New account** blade, specify the configuration that you want for the Azure Cosmos DB account.

   With Azure Cosmos DB, you can choose one of four programming models: Gremlin (graph), MongoDB, SQL (DocumentDB), and Table (key-value).

   In this quick-start article we program against the DocumentDB API, so choose **SQL (DocumentDB)** as you fill out the form. But if you have graph data for a social media app, or key/value (table) data, or data migrated from a MongoDB app, realize that Azure Cosmos DB can provide a highly available, globally distributed database service platform for all your mission-critical applications.

   Complete the fields on the **New account** blade, using the information in the following screenshot as a guide. When you set up your account, choose unique values that do not match those in the screenshot.



Azure Cosmos DB
New account

* ID
Enter account ID
documents.azure.com

API
Please choose an API

* Subscription
Visual Studio Ultimate with MSDN

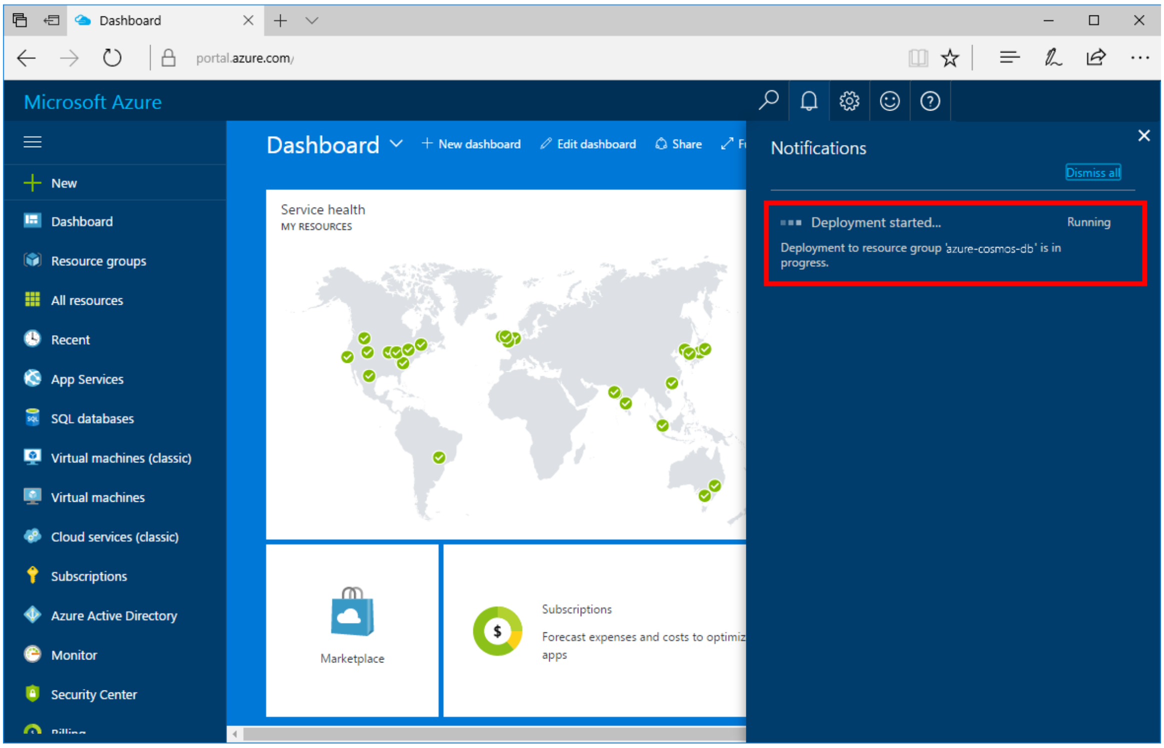
* Resource Group
● Create new ○ Use existing

* Location
North Central US

Create
Automation options

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	*Unique value*	A unique name that identifies your Azure Cosmos DB account. The string *documents.azure.com* is appended to the ID you provide to create your URI, so use a unique but identifiable ID. The ID can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain from 3 through 50 characters.
API	SQL (DocumentDB)	We program against the DocumentDB API later in this article.
Subscription	*Your subscription*	The Azure subscription that you want to use for your Azure Cosmos DB account.
Resource Group	*The same value as ID*	The new resource-group name for your account. For simplicity, you can use the same name as your ID.
Location	*The region closest to your users*	The geographic location in which to host your Azure Cosmos DB account. Choose the location that's closest to your users to give them the fastest access to the data.

4. Click **Create** to create the account.
5. On the top toolbar, click **Notifications** to monitor the deployment process.

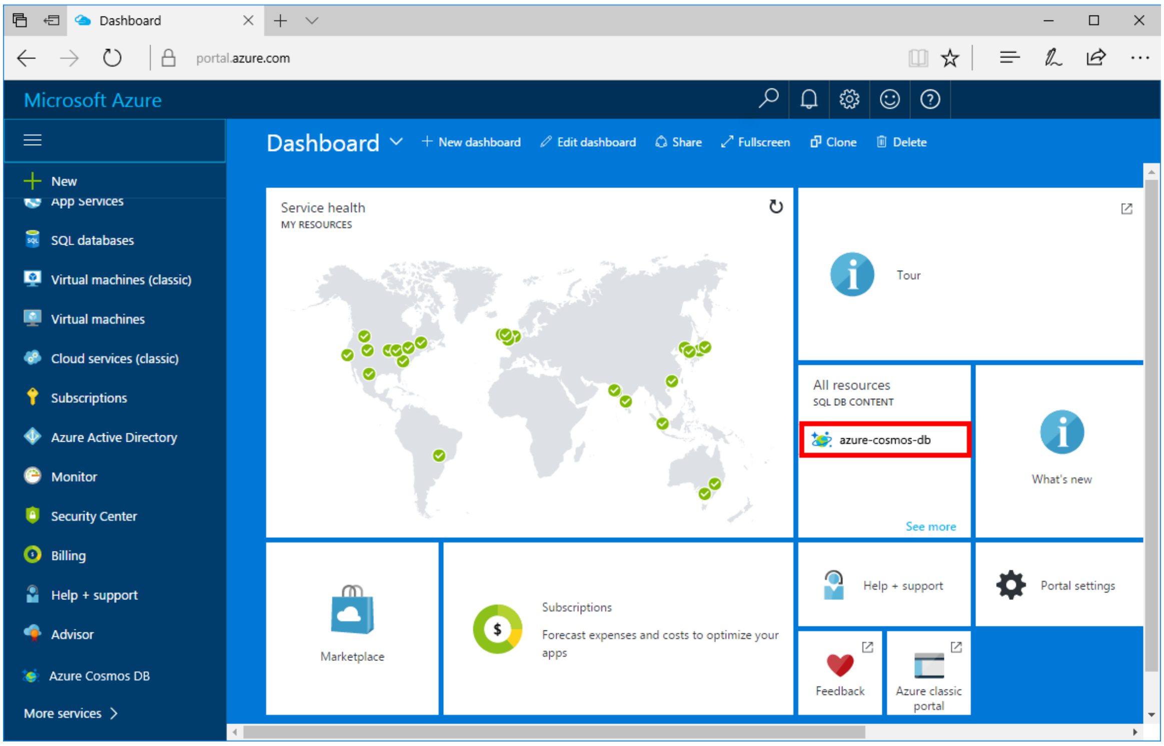


Notifications

Deployment started... Running

Deployment to resource group 'azure-cosmos-db' is in progress.

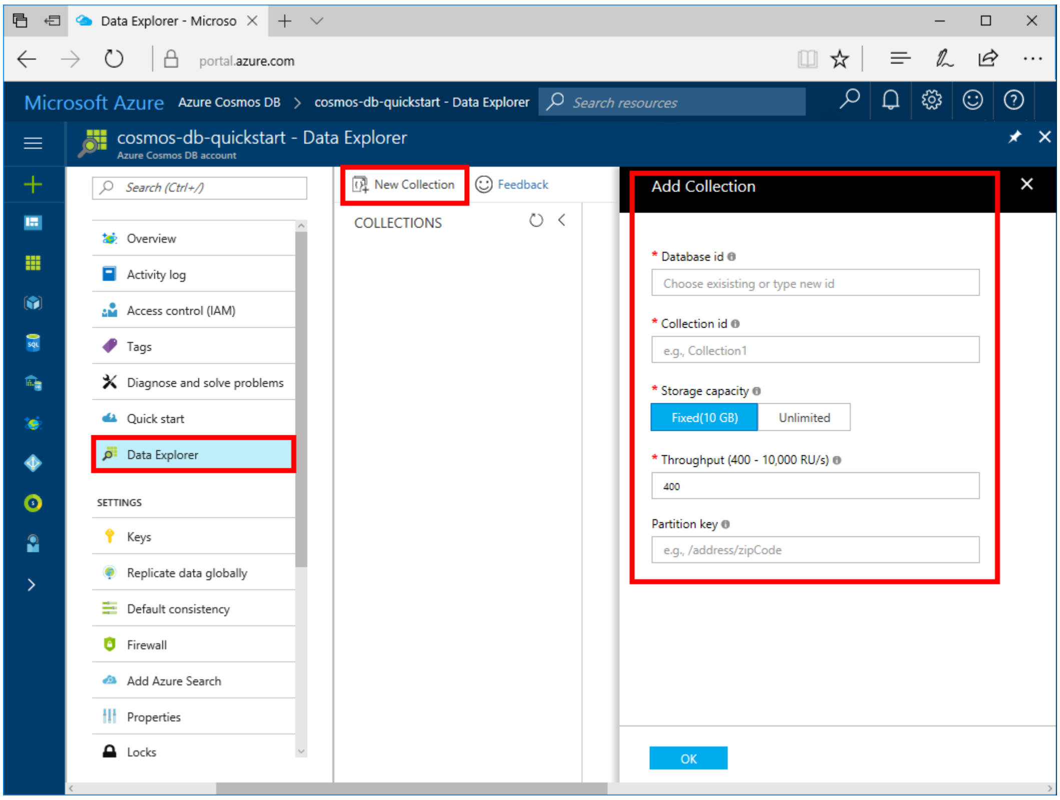
6. When the deployment is complete, open the new account from the **All Resources** tile.



Add a collection

You can now use Data Explorer to create a collection and add data to your database.

1. In the Azure portal, in the left pane, click **Data Explorer**.
2. On the **Data Explorer** blade, click **New Collection**, and then provide the following information:



SETTING	SUGGESTED VALUE	DESCRIPTION
Database id	Items	The ID for your new database. Database names must contain from 1 through 255 characters, and they cannot contain /, \, #, ?, or a trailing space.
Collection id	ToDoList	The ID for your new collection. Collection names have the same character requirements as database IDs.
Storage capacity	Fixed (10 GB)	Use the default value. This is the storage capacity of the database.
Throughput	400 RU	Use the default value. If you want to reduce latency, you can scale up the throughput later.

SETTING	SUGGESTED VALUE	DESCRIPTION
Partition key	/userid	A partition key that distributes data evenly to each partition. Selecting the correct partition key is important in creating a performant collection. To learn more, see Designing for partitioning.

3. After you've completed the form, click **OK**.

Get Apache TinkerPop

Get Apache TinkerPop by doing the following:

1. Remote to the master node of the HDInsight cluster `ssh tinkerpop3-cosmosdb-demo-ssh.azurehdinsight.net`.
2. Clone the TinkerPop3 source code, build it locally, and install it to Maven cache.

```
git clone https://github.com/apache/tinkerpop.git
cd tinkerpop
mvn clean install
```

3. Install the Spark-Gremlin plug-in

   a. The installation of the plug-in is handled by Grape. Populate the repositories information for Grape so it can download the plug-in and its dependencies.

   Create the grape configuration file if it's not present at `~/.groovy/grapeConfig.xml`. Use the following settings:

```
<ivysettings>
<settings defaultResolver="downloadGrapes"/>
<resolvers>
  <chain name="downloadGrapes">
  <filesystem name="cachedGrapes">
    <ivy pattern="${user.home}/.groovy/grapes/[organisation]/[module]/ivy-[revision].xml"/>
    <artifact pattern="${user.home}/.groovy/grapes/[organisation]/[module]/[type]s/[artifact]-[revision].[ext]"/>
  </filesystem>
  <ibiblio name="codehaus" root="http://repository.codehaus.org/" m2compatible="true"/>
  <ibiblio name="central" root="http://central.maven.org/maven2/" m2compatible="true"/>
  <ibiblio name="jitpack" root="https://jitpack.io" m2compatible="true"/>
  <ibiblio name="java.net2" root="http://download.java.net/maven/2/" m2compatible="true"/>
  <ibiblio name="apache-snapshots" root="http://repository.apache.org/snapshots/" m2compatible="true"/>
  <ibiblio name="local" root="file:${user.home}/.m2/repository/" m2compatible="true"/>
  </chain>
</resolvers>
</ivysettings>
```

   b. Start Gremlin console `bin/gremlin.sh`.

   c. Install the Spark-Gremlin plug-in with version 3.3.0-SNAPSHOT, which you built in the previous steps:

```
$ bin/gremlin.sh

       \,,,/
       (o o)
-----oOOo-(3)-oOOo-----
plugin activated: tinkerpop.server
plugin activated: tinkerpop.utilities
plugin activated: tinkerpop.tinkergraph
gremlin> :install org.apache.tinkerpop spark-gremlin 3.3.0-SNAPSHOT
==>loaded: [org.apache.tinkerpop, spark-gremlin, 3.3.0-SNAPSHOT] - restart the console to use [tinkerpop.spark]
gremlin> :q
$ bin/gremlin.sh

       \,,,/
       (o o)
-----oOOo-(3)-oOOo-----
plugin activated: tinkerpop.server
plugin activated: tinkerpop.utilities
plugin activated: tinkerpop.tinkergraph
gremlin> :plugin use tinkerpop.spark
==>tinkerpop.spark activated
```

4. Check to see whether `Hadoop-Gremlin` is activated with `:plugin list`. Disable this plug-in, because it could interfere with the Spark-Gremlin plug-in `:plugin unuse tinkerpop.hadoop`.

Prepare TinkerPop3 dependencies

When you built TinkerPop3 in the previous step, the process also pulled all jar dependencies for Spark and Hadoop in the target directory. Use the jars that are pre-installed with HDI, and pull in additional dependencies only as necessary.

1. Go to the Gremlin Console target directory at `tinkerpop/gremlin-console/target/apache-tinkerpop-gremlin-console-3.3.0-SNAPSHOT-standalone`.
2. Move all jars under `ext/` to `lib/`: `find ext/ -name '*.jar' -exec mv {} lib/ \;`.
3. Remove all jar libraries under `lib/` that are not in the following list:

```
# TinkerPop3
gremlin-console-3.3.0-SNAPSHOT.jar
gremlin-core-3.3.0-SNAPSHOT.jar
gremlin-groovy-3.3.0-SNAPSHOT.jar
gremlin-shaded-3.3.0-SNAPSHOT.jar
hadoop-gremlin-3.3.0-SNAPSHOT.jar
spark-gremlin-3.3.0-SNAPSHOT.jar
tinkergraph-gremlin-3.3.0-SNAPSHOT.jar

# Gremlin depedencies
asm-3.2.jar
avro-1.7.4.jar
caffeine-2.3.1.jar
cglib-2.2.1-v20090111.jar
gbench-0.4.3-groovy-2.4.jar
gprof-0.3.1-groovy-2.4.jar
groovy-2.4.9-indy.jar
groovy-2.4.9.jar
groovy-console-2.4.9.jar
groovy-groovysh-2.4.9-indy.jar
groovy-json-2.4.9-indy.jar
groovy-jsr223-2.4.9-indy.jar
groovy-sql-2.4.9-indy.jar
groovy-swing-2.4.9.jar
groovy-templates-2.4.9.jar
groovy-xml-2.4.9.jar
hadoop-yarn-server-nodemanager-2.7.2.jar
hppc-0.7.1.jar
javatuples-1.2.jar
jaxb-impl-2.2.3-1.jar
jbcrypt-0.4.jar
jcabi-log-0.14.jar
jcabi-manifests-1.1.jar
jersey-core-1.9.jar
jersey-guice-1.9.jar
jersey-json-1.9.jar
jettison-1.1.jar
scalatest_2.11-2.2.6.jar
servlet-api-2.5.jar
snakeyaml-1.15.jar
unused-1.0.0.jar
xml-apis-1.3.04.jar
```

Get the Azure Cosmos DB Spark connector

1. Get the Azure Cosmos DB Spark connector `azure-documentdb-spark-0.0.3-SNAPSHOT.jar` and Cosmos DB Java SDK `azure-documentdb-1.10.0.jar` from Azure Cosmos DB Spark Connector on GitHub.

2. Alternatively, you can build it locally. Because the latest version of Spark-Gremlin was built with Spark 1.6.1 and is not compatible with Spark 2.0.2, which is currently used in the Azure Cosmos DB Spark connector, you can build the latest TinkerPop3 code and install the jars manually. Do the following:

   a. Clone the Azure Cosmos DB Spark connector.

   b. Build TinkerPop3 (already done in previous steps). Install all TinkerPop 3.3.0-SNAPSHOT jars locally.

```
mvn install:install-file -Dfile="gremlin-core-3.3.0-SNAPSHOT.jar" -DgroupId=org.apache.tinkerpop -DartifactId=gremlin-core -Dversion=3.3.0-SNAPSHOT -Dpackaging=jar
mvn install:install-file -Dfile="gremlin-groovy-3.3.0-SNAPSHOT.jar" -DgroupId=org.apache.tinkerpop -DartifactId=gremlin-groovy -Dversion=3.3.0-SNAPSHOT -Dpackaging=jar`
mvn install:install-file -Dfile="gremlin-shaded-3.3.0-SNAPSHOT.jar" -DgroupId=org.apache.tinkerpop -DartifactId=gremlin-shaded -Dversion=3.3.0-SNAPSHOT -Dpackaging=jar`
mvn install:install-file -Dfile="hadoop-gremlin-3.3.0-SNAPSHOT.jar" -DgroupId=org.apache.tinkerpop -DartifactId=hadoop-gremlin -Dversion=3.3.0-SNAPSHOT -Dpackaging=jar`
mvn install:install-file -Dfile="spark-gremlin-3.3.0-SNAPSHOT.jar" -DgroupId=org.apache.tinkerpop -DartifactId=spark-gremlin -Dversion=3.3.0-SNAPSHOT -Dpackaging=jar`
mvn install:install-file -Dfile="tinkergraph-gremlin-3.3.0-SNAPSHOT.jar" -DgroupId=org.apache.tinkerpop -DartifactId=tinkergraph-gremlin -Dversion=3.3.0-SNAPSHOT -Dpackaging=jar`
```

   c. Update `tinkerpop.version` `azure-documentdb-spark/pom.xml` to `3.3.0-SNAPSHOT`.

   d. Build with Maven. The needed jars are placed in `target` and `target/alternateLocation`.

```
git clone https://github.com/Azure/azure-cosmosdb-spark.git
cd azure-documentdb-spark
mvn clean package
```

3. Copy the previously mentioned jars to a local directory at ~/azure-documentdb-spark:

```
$ azure-documentdb-spark:
mkdir ~/azure-documentdb-spark
cp target/azure-documentdb-spark-0.0.3-SNAPSHOT.jar ~/azure-documentdb-spark
cp target/alternateLocation/azure-documentdb-1.10.0.jar ~/azure-documentdb-spark
```

Distribute the dependencies to the Spark worker nodes

1. Because the transformation of graph data depends on TinkerPop3, you must distribute the related dependencies to all Spark worker nodes.

2. Copy the previously mentioned Gremlin dependencies, the CosmosDB Spark connector jar, and CosmosDB Java SDK to the worker nodes by doing the following:

   a. Copy all the jars into `~/azure-documentdb-spark`.

```
$ /home/sshuser/tinkerpop/gremlin-console/target/apache-tinkerpop-gremlin-console-3.3.0-SNAPSHOT-standalone:
cp lib/* ~/azure-documentdb-spark
```

   b. Get the list of all Spark worker nodes, which you can find on Ambari Dashboard, in the `Spark2 Clients` list in the `Spark2` section.

   c. Copy that directory to each of the nodes.

```
scp -r ~/azure-documentdb-spark sshuser@wn0-cosmos:/home/sshuser
scp -r ~/azure-documentdb-spark sshuser@wn1-cosmos:/home/sshuser
...
```

Set up the environment variables

1. Find the HDP version of the Spark cluster. It is the directory name under `/usr/hdp/` (for example, 2.5.4.2-7).

2. Set hdp.version for all nodes. In Ambari Dashboard, go to **YARN section** > **Configs** > **Advanced**, and then do the following:

   a. In `Custom yarn-site`, add a new property `hdp.version` with the value of the HDP version on the master node.

   b. Save the configurations. There are warnings, which you can ignore.

   c. Restart the YARN and Oozie services as the notification icons indicate.

3. Set the following environment variables on the master node (replace the values as appropriate):

```
export HADOOP_GREMLIN_LIBS=/home/sshuser/tinkerpop/gremlin-console/target/apache-tinkerpop-gremlin-console-3.3.0-SNAPSHOT-standalone/ext/spark-gremlin/lib
export CLASSPATH=$CLASSPATH:$HADOOP_CONF_DIR:/usr/hdp/current/spark2-client/jars/*:/home/sshuser/azure-documentdb-spark/*
export HDP_VERSION=2.5.4.2-7
export HADOOP_HOME=${HADOOP_HOME:-/usr/hdp/current/hadoop-client}
```

Prepare the graph configuration

1. Create a configuration file with the Azure Cosmos DB connection parameters and Spark settings, and put it at `tinkerpop/gremlin-console/target/apache-tinkerpop-gremlin-console-3.3.0-SNAPSHOT-standalone/conf/hadoop/gremlin-spark.properties`.

```
gremlin.graph=org.apache.tinkerpop.gremlin.hadoop.structure.HadoopGraph
gremlin.hadoop.jarsInDistributedCache=true
gremlin.hadoop.defaultGraphComputer=org.apache.tinkerpop.gremlin.spark.process.computer.SparkGraphComputer

gremlin.hadoop.graphReader=com.microsoft.azure.documentdb.spark.gremlin.DocumentDBInputRDD
gremlin.hadoop.graphWriter=com.microsoft.azure.documentdb.spark.gremlin.DocumentDBOutputRDD

####################################
# SparkGraphComputer Configuration #
####################################
spark.master=yarn
spark.executor.memory=3g
spark.executor.instances=6
spark.serializer=org.apache.spark.serializer.KryoSerializer
spark.kryo.registrator=org.apache.tinkerpop.gremlin.spark.structure.io.gryo.GryoRegistrator
gremlin.spark.persistContext=true

# Classpath for the driver and executors
spark.driver.extraClassPath=/usr/hdp/current/spark2-client/jars/*:/home/sshuser/azure-documentdb-spark/*
spark.executor.extraClassPath=/usr/hdp/current/spark2-client/jars/*:/home/sshuser/azure-documentdb-spark/*

######################################
# DocumentDB Spark connector         #
######################################
spark.documentdb.connectionMode=Gateway
spark.documentdb.schema_samplingratio=1.0
spark.documentdb.Endpoint=https://FILLIN.documents.azure.com:443/
spark.documentdb.Masterkey=FILLIN
spark.documentdb.Database=FILLIN
spark.documentdb.Collection=FILLIN
spark.documentdb.preferredRegions=FILLIN
```

2. Update the `spark.driver.extraClassPath` and `spark.executor.extraClassPath` to include the directory of the jars that you distributed in the previous step, in this case `/home/sshuser/azure-documentdb-spark/*`.

3. Provide the following details for Azure Cosmos DB:

```
spark.documentdb.Endpoint=https://FILLIN.documents.azure.com:443/
spark.documentdb.Masterkey=FILLIN
spark.documentdb.Database=FILLIN
spark.documentdb.Collection=FILLIN
# Optional
#spark.documentdb.preferredRegions=West\ US;West\ US\ 2
```

Load the TinkerPop graph, and save it to Azure Cosmos DB

To demonstrate how to persist a graph into Azure Cosmos DB, this example uses the TinkerPop predefined TinkerPop modern graph. The graph is stored in Kryo format, and it's provided in the TinkerPop repository.

1. Because you are running Gremlin in YARN mode, you must make the graph data available in the Hadoop file system. Use the following commands to make a directory and copy the local graph file into it.

```
$ tinkerpop:
hadoop fs -mkdir /graphData
hadoop fs -copyFromLocal ~/tinkerpop/data/tinkerpop-modern.kryo /graphData/tinkerpop-modern.kryo
```

2. Temporarily update the `gremlin-spark.properties` file to use `GryoInputFormat` to read the graph. Also indicate `inputLocation` as the directory you create, as in the following:

```
gremlin.hadoop.graphReader=org.apache.tinkerpop.gremlin.hadoop.structure.io.gryo.GryoInputFormat
gremlin.hadoop.inputLocation=/graphData/tinkerpop-modern.kryo
```

3. Start Gremlin Console, and then create the following computation steps to persist data to the configured Azure Cosmos DB collection:

   a. Create the graph `graph = GraphFactory.open("conf/hadoop/gremlin-spark.properties")`.

   b. Use SparkGraphComputer for writing

```
graph.compute(SparkGraphComputer.class).result(GraphComputer.ResultGraph.NEW).persist(GraphComputer.Persist.EDGES).program(TraversalVertexProgram.build().traversal(graph.traversal().withComputer(Computer.compute(SparkGraphCom
groovy","g.V()").create(graph)).submit().get()
```

.

```
gremlin> graph = GraphFactory.open("conf/hadoop/gremlin-spark.properties")
==>hadoopgraph[gryoinputformat->documentdboutputrdd]
gremlin> hg = graph.
            compute(SparkGraphComputer.class).
            result(GraphComputer.ResultGraph.NEW).
            persist(GraphComputer.Persist.EDGES).
            program(TraversalVertexProgram.build().
                traversal(graph.traversal().withComputer(Computer.compute(SparkGraphComputer.class)), "gremlin-groovy", "g.V()").
                create(graph)).
            submit().
            get()
==>result[hadoopgraph[documentdbinputrdd->documentdboutputrdd],memory[size:1]]
```

4. From Data Explorer, you can verify that the data has been persisted to Azure Cosmos DB.

Load the graph from Azure Cosmos DB, and run Gremlin queries

1. To load the graph, edit `gremlin-spark.properties` to set `graphReader` to `DocumentDBInputRDD`:

```
gremlin.hadoop.graphReader=com.microsoft.azure.documentdb.spark.gremlin.DocumentDBInputRDD
```

2. Load the graph, traverse the data, and run Gremlin queries with it by doing the following:

   a. Start the Gremlin Console `bin/gremlin.sh`.

   b. Create the graph with the configuration `graph = GraphFactory.open('conf/hadoop/gremlin-spark.properties')`.

   c. Create a graph traversal with SparkGraphComputer `g = graph.traversal().withComputer(SparkGraphComputer)`.

   d. Run the following Gremlin graph queries:

```
gremlin> graph = GraphFactory.open("conf/hadoop/gremlin-spark.properties")
==>hadoopgraph[documentdbinputrdd->documentdboutputrdd]
gremlin> g = graph.traversal().withComputer(SparkGraphComputer)
==>graphtraversalsource[hadoopgraph[documentdbinputrdd->documentdboutputrdd], sparkgraphcomputer]
gremlin> g.V().count()
==>6
gremlin> g.E().count()
==>6
gremlin> g.V(1).out().values('name')
==>josh
==>vadas
==>lop
gremlin> g.V().hasLabel('person').coalesce(values('nickname'), values('name'))
==>josh
==>peter
==>vadas
==>marko
gremlin> g.V().hasLabel('person').
          choose(values('name')).
            option('marko', values('age')).
            option('josh', values('name')).
            option('vadas', valueMap()).
            option('peter', label())
==>josh
==>person
==>[name:[vadas],age:[27]]
==>29
```

NOTE

To see more detailed logging, set the log level in `conf/log4j-console.properties` to a more verbose level.

Next steps

In this quick-start article, you've learned how to work with graphs by combining Azure Cosmos DB and Spark.

Deploy Azure Cosmos DB and Azure App Service Web Apps using an Azure Resource Manager Template

6/6/2017 • 6 min to read • Edit Online

This tutorial shows you how to use an Azure Resource Manager template to deploy and integrate Microsoft Azure Cosmos DB, 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 Azure Cosmos DB 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 an Azure Cosmos DB account and a web app in Azure App Service?
- How can I use an Azure Resource Manager template to deploy and integrate an Azure Cosmos DB 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 an Azure Cosmos DB account, Web Apps, and deploy a demo application sample template to a local folder (e.g. C:\Azure Cosmos DBTemplates). This template will deploy an Azure Cosmos DB account, an App Service web app, and a web application. It will also automatically configure the web application to connect to the Azure Cosmos DB account.
2. Download the Create an Azure Cosmos DB account and Web Apps sample template to a local folder (e.g. C:\Azure Cosmos DBTemplates). This template will deploy an Azure Cosmos DB account, an App Service web app, and will modify the site's application settings to easily surface Azure Cosmos DB connection information, but does not include a web application.

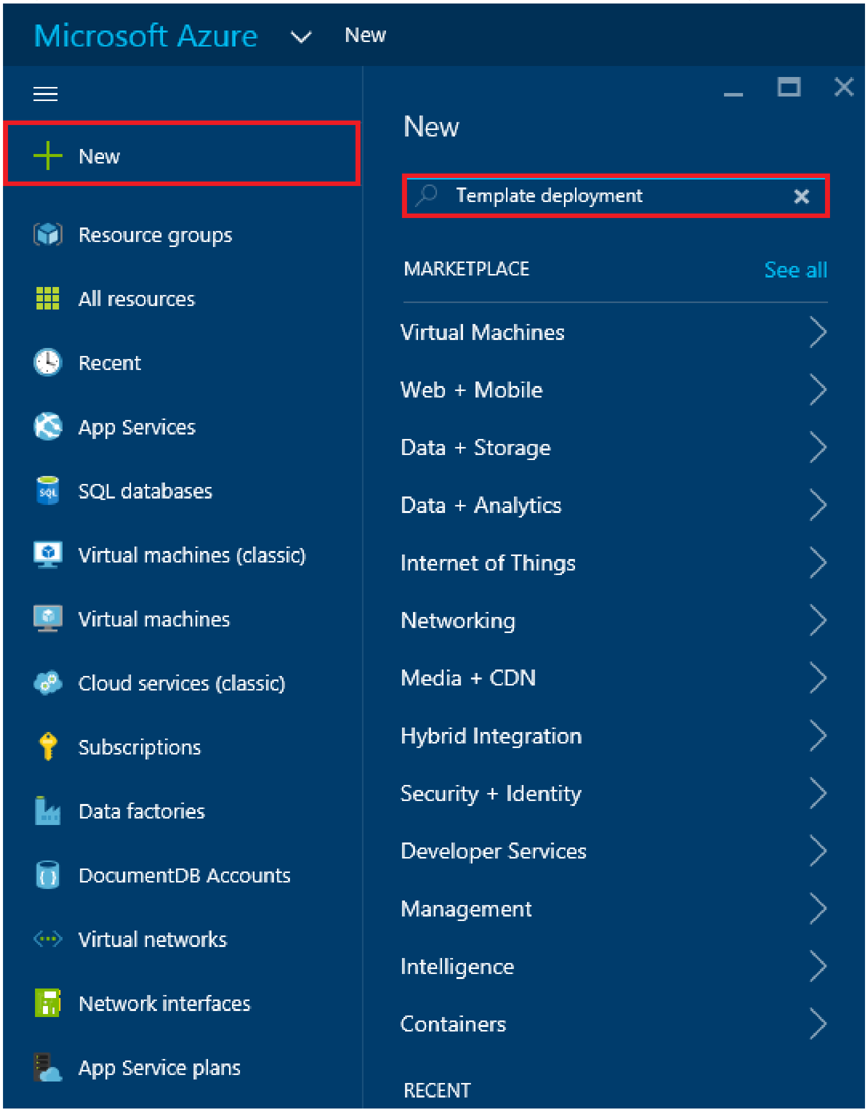
Step 2: Deploy the Azure Cosmos DB account, App Service web app and demo application sample

Now let's deploy our first template.

TIP

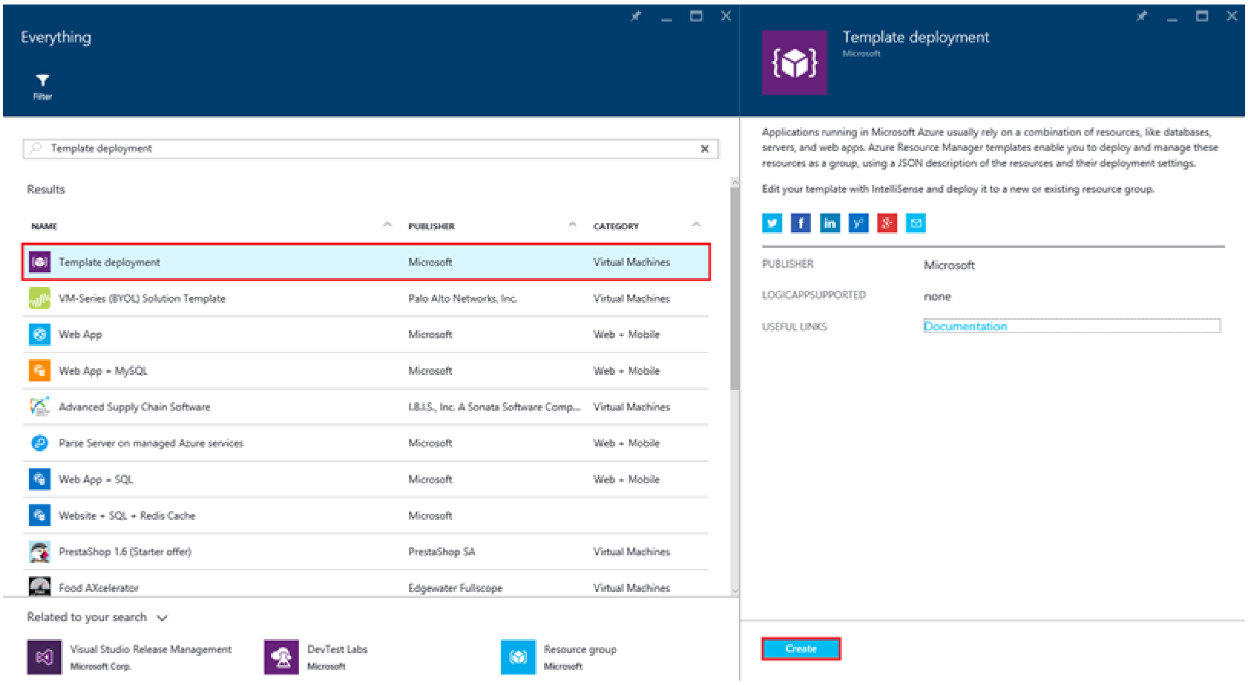
The template does not validate that the web app name and Azure Cosmos DB account name entered below are a) valid and b) available. It is highly recommended that you verify the availability of the names you plan to supply prior to submitting the deployment.

1. Login to the Azure Portal, click New and search for "Template deployment".



Template deployment

2. Select the Template deployment item and click **Create**



Template deployment

Results

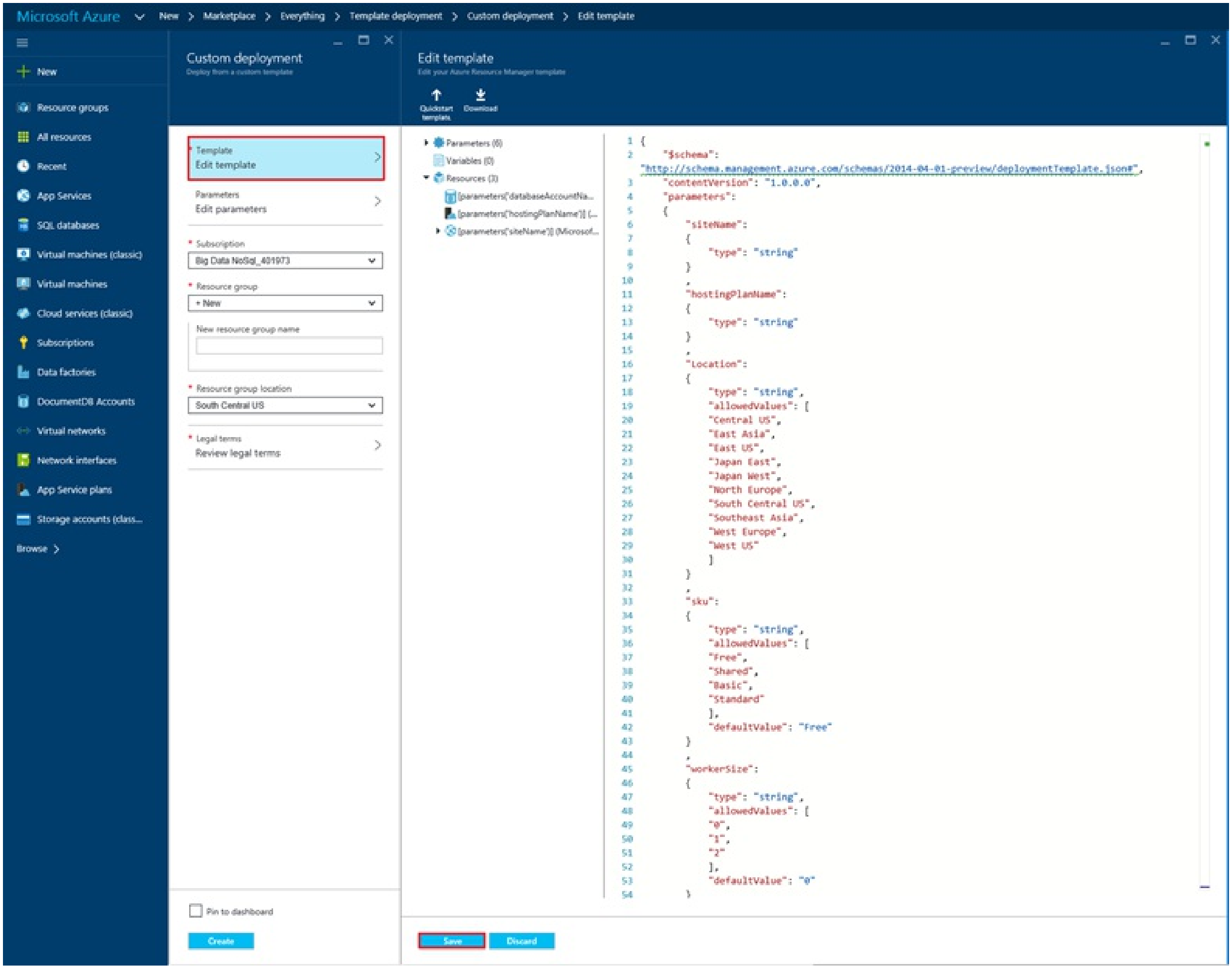
NAME	PUBLISHER	CATEGORY
Template deployment	Microsoft	Virtual Machines
VM-Series (BYOL) Solution Template	Palo Alto Networks, Inc.	Virtual Machines
Web App	Microsoft	Web + Mobile
Web App + MySQL	Microsoft	Web + Mobile
Advanced Supply Chain Software	I.B.I.S., Inc. A Sonata Software Comp...	Virtual Machines
Parse Server on managed Azure services	Microsoft	Web + Mobile
Web App + SQL	Microsoft	Web + Mobile
Website + SQL + Redis Cache	Microsoft	
PrestaShop 1.6 (Starter offer)	PrestaShop SA	Virtual Machines
Food AXcelerator	Edgewater Fullscope	Virtual Machines

Applications running in Microsoft Azure usually rely on a combination of resources, like databases, servers, and web apps. Azure Resource Manager templates enable you to deploy and manage these resources as a group, using a JSON description of the resources and their deployment settings.

Edit your template with IntelliSense and deploy it to a new or existing resource group.

PUBLISHER	Microsoft
LOGICAPPSUPPORTED	none
USEFUL LINKS	Documentation

3. Click **Edit template**, paste the contents of the DocDBWebsiteTodo.json template file, and click **Save**.



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

```
{
    "$schema":
"http://schema.management.azure.com/schemas/2014-04-01-preview/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters":
    {
        "siteName":
        {
            "type": "string"
        }
        ,
        "hostingPlanName":
        {
            "type": "string"
        }
        ,
        "location":
        {
            "type": "string",
            "allowedValues": [
            "Central US",
            "East Asia",
            "East US",
            "Japan East",
            "Japan West",
            "North Europe",
            "South Central US",
            "Southeast Asia",
            "West Europe",
            "West US"
            ]
        }
        ,
        "sku":
        {
            "type": "string",
            "allowedValues": [
            "Free",
            "Shared",
            "Basic",
            "Standard"
            ],
            "defaultValue": "Free"
        }
        ,
        "workerSize":
        {
            "type": "string",
            "allowedValues": [
            "0",
            "1",
            "2"
            ],
            "defaultValue": "0"
        }
```

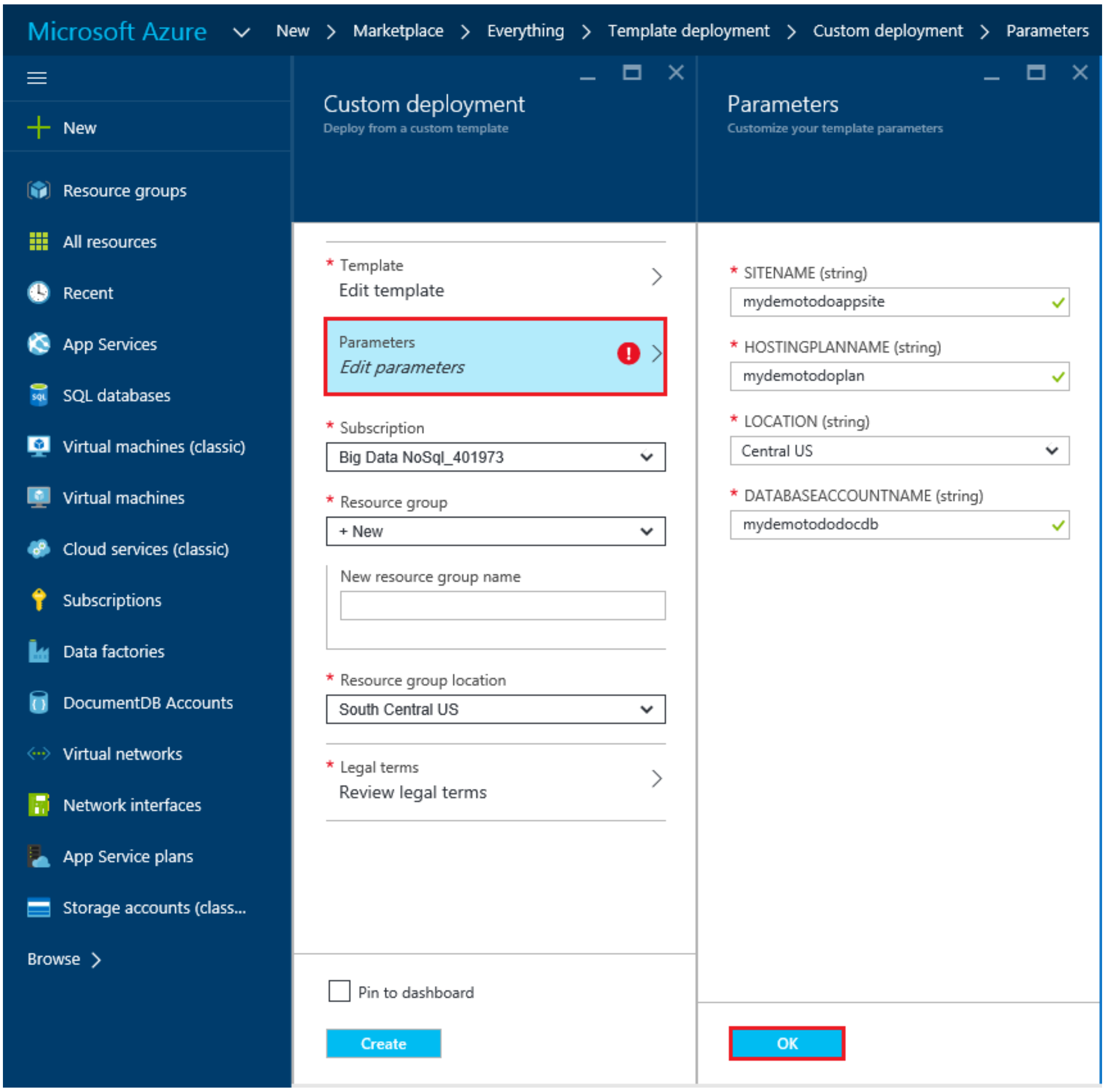
4. Click **Edit parameters**, provide values for each of the mandatory parameters, and click **OK**. The parameters are as follows:

   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 Azure Cosmos DB and web app resources.

   d. DATABASEACCOUNTNAME: Specifies the name of the Azure Cosmos DB account to create.



* 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

* SITENAME (string)
mydemotodoappsite ✓

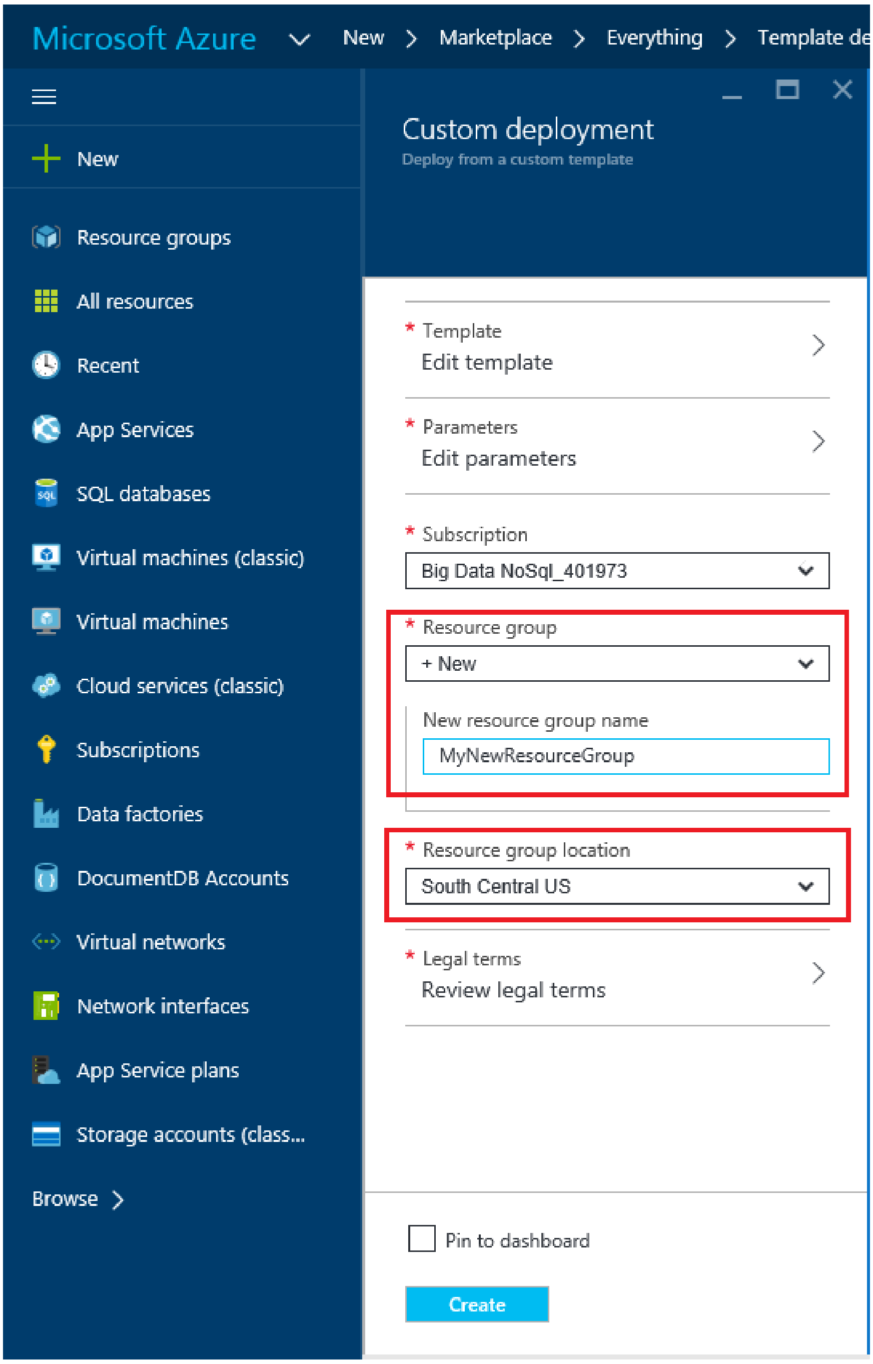
* HOSTINGPLANNAME (string)
mydemotodoplan ✓

* LOCATION (string)
Central US

* DATABASEACCOUNTNAME (string)
mydemotododocdb ✓

OK

5. Choose an existing Resource group or provide a name to make a new resource group, and choose a location for the resource group.



Custom deployment
Deploy from a custom template

* Template
Edit template

* Parameters
Edit parameters

* Subscription
Big Data NoSql_401973

* Resource group
+ New

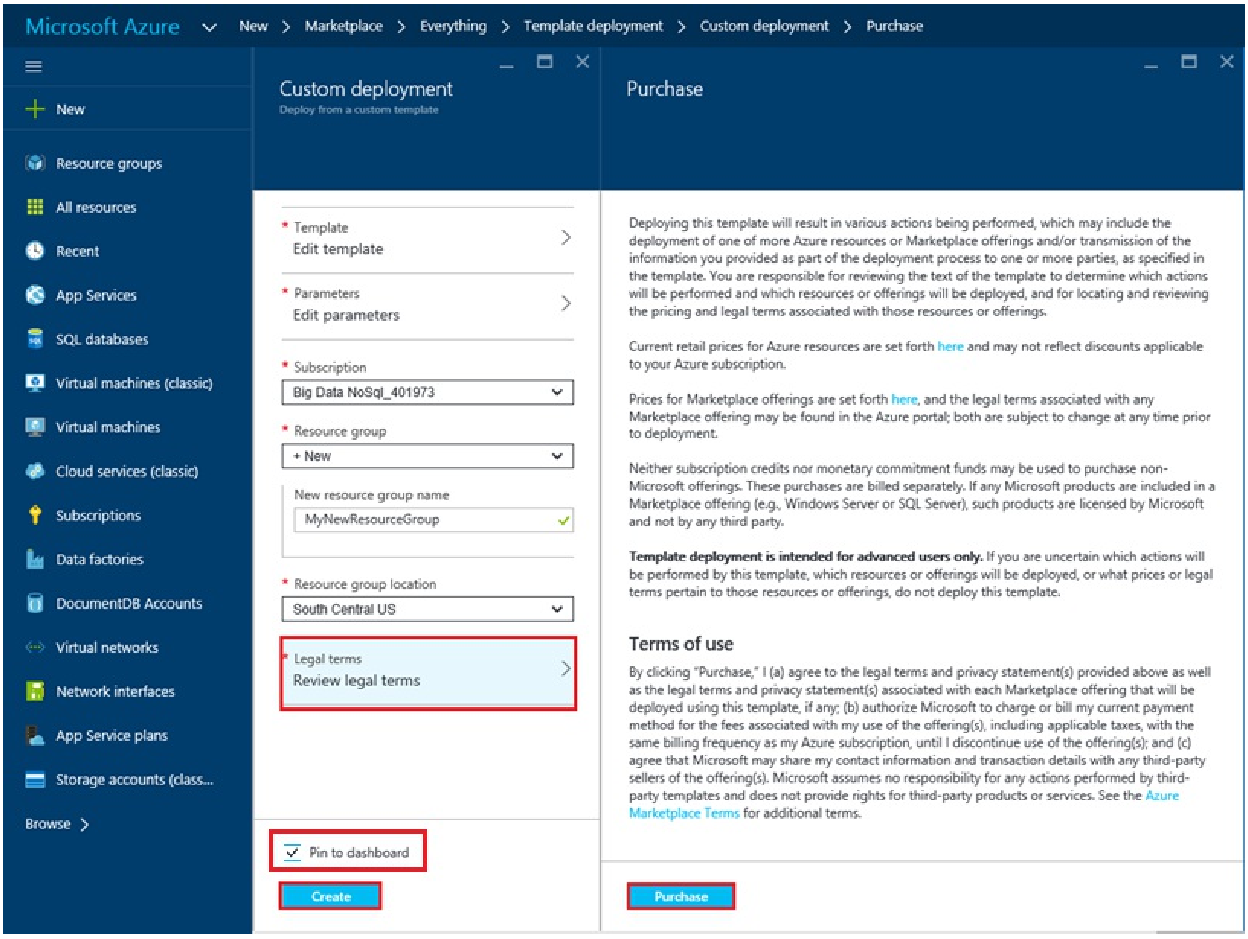
New resource group name
MyNewResourceGroup

* Resource group location
South Central US

* Legal terms
Review legal terms

□ Pin to dashboard

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.



* 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

Deploying this template will result in various actions being performed, which may include the deployment of one of more Azure resources or Marketplace offerings and/or transmission of the information you provided as part of the deployment process to one or more parties, as specified in the template. You are responsible for reviewing the text of the template to determine which actions will be performed and which resources or offerings will be deployed, and for locating and reviewing the pricing and legal terms associated with those resources or offerings.

Current retail prices for Azure resources are set forth here and may not reflect discounts applicable to your Azure subscription.

Prices for Marketplace offerings are set forth here, and the legal terms associated with any Marketplace offering may be found in the Azure portal; both are subject to change at any time prior to deployment.

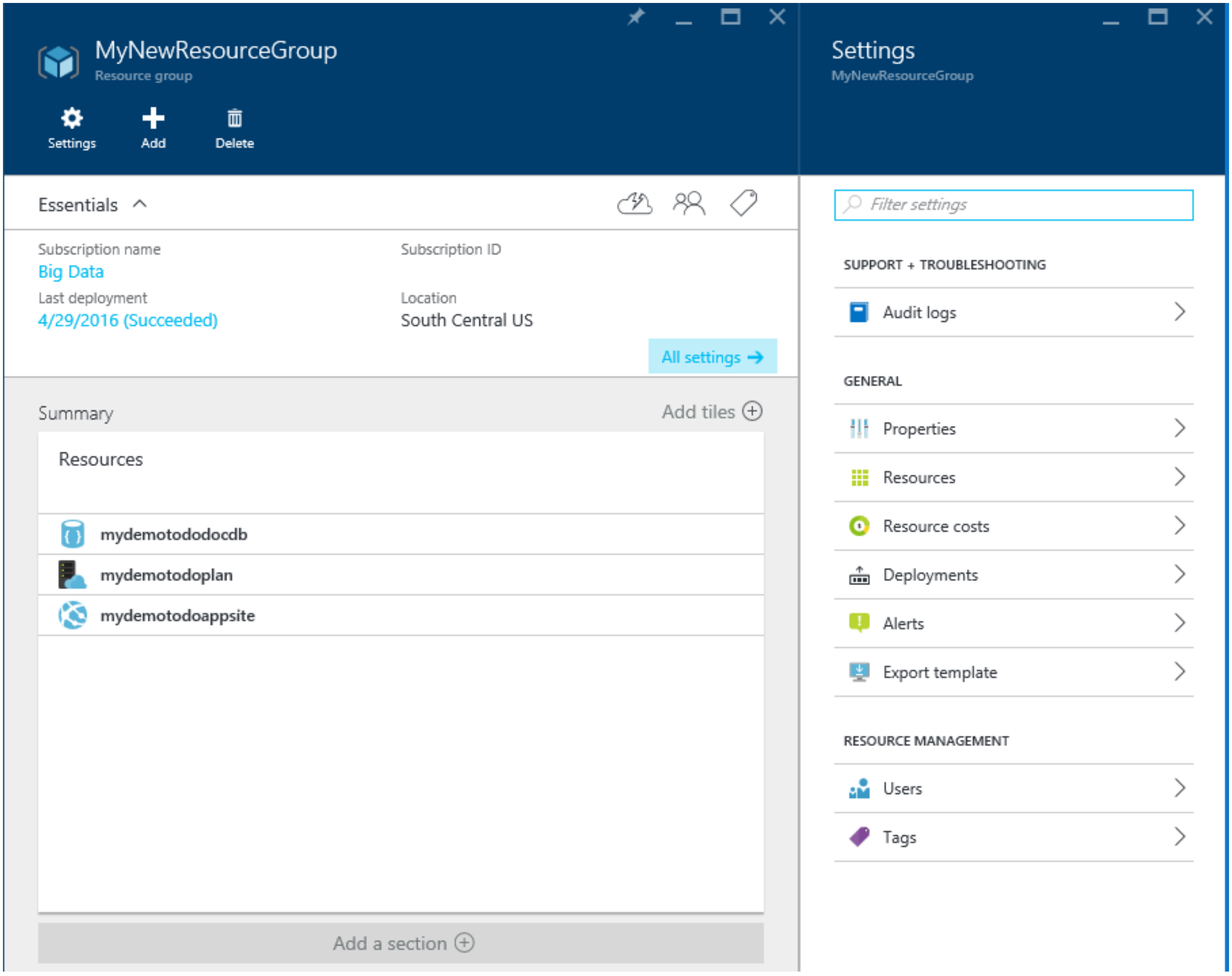
Neither subscription credits nor monetary commitment funds may be used to purchase non-Microsoft offerings. These purchases are billed separately. If any Microsoft products are included in a Marketplace offering (e.g., Windows Server or SQL Server), such products are licensed by Microsoft and not by any third party.

Template deployment is intended for advanced users only. If you are uncertain which actions will be performed by this template, which resources or offerings will be deployed, or what prices or legal terms pertain to those resources or offerings, do not deploy this template.

Terms of use

By clicking "Purchase," I (a) agree to the legal terms and privacy statement(s) provided above as well as the legal terms and privacy statement(s) associated with each Marketplace offering that will be deployed using this template, if any; (b) authorize Microsoft to charge or bill my current payment method for the fees associated with my use of the offering(s), including applicable taxes, with the same billing frequency as my Azure subscription, until I discontinue use of the offering(s); and (c) agree that Microsoft may share my contact information and transaction details with any third-party sellers of the offering(s). Microsoft assumes no responsibility for any actions performed by third-party templates and does not provide rights for third-party products or services. See the Azure Marketplace Terms for additional terms.

7. When the deployment finishes, the Resource group blade will open.



Subscription name: Big Data

Subscription ID

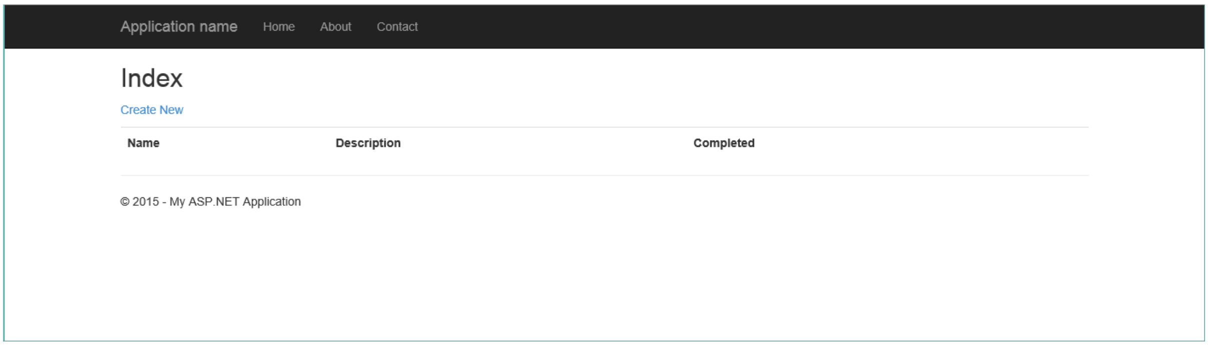
Last deployment: 4/29/2016 (Succeeded)

Location: South Central US

Resources

- mydemotododocdb
- mydemotodoplan
- mydemotodoappsite

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:



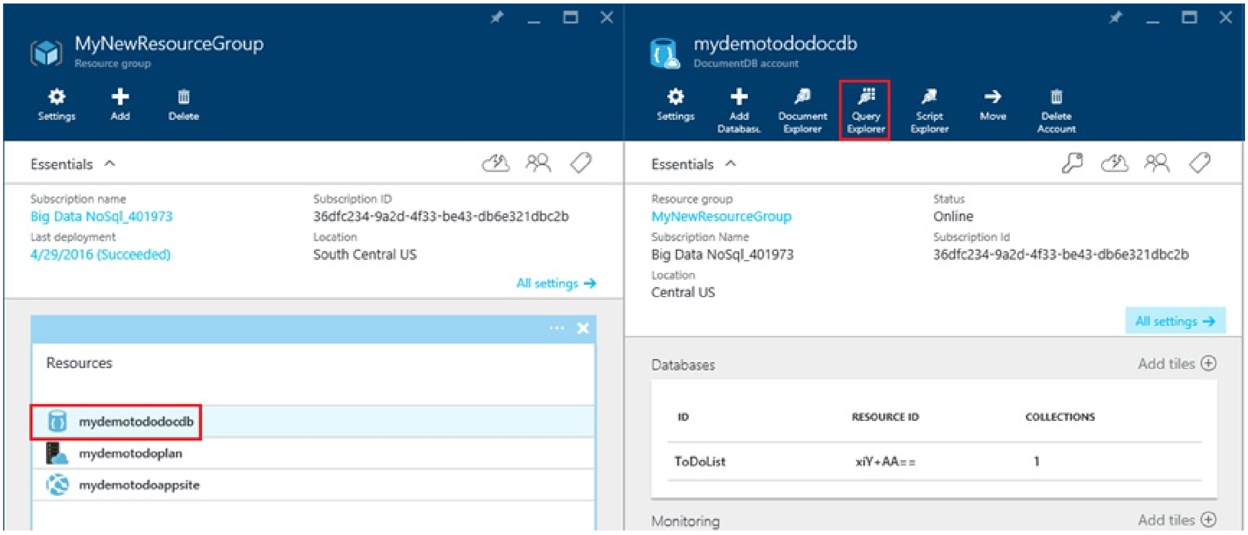
Index

Create New

Name	Description	Completed

© 2015 - My ASP.NET 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 Azure Cosmos DB account resource in the Resources list and then click **Query Explorer**.

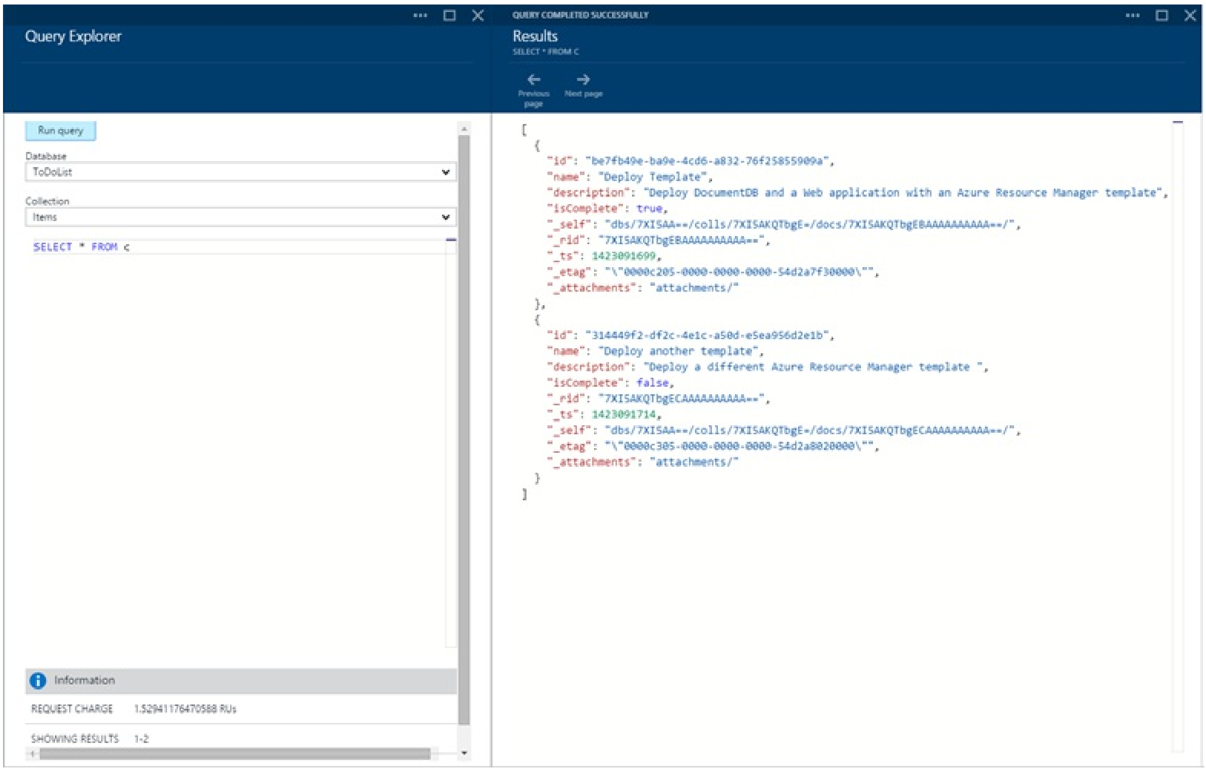


Subscription name: Big Data NoSql_401973
Subscription ID: 36dfc234-9a2d-4f33-be43-db6e321dbc2b
Last deployment: 4/29/2016 (Succeeded)
Location: South Central US

Resource group: MyNewResourceGroup
Status: Online
Subscription Name: Big Data NoSql_401973
Subscription Id: 36dfc234-9a2d-4f33-be43-db6e321dbc2b
Location: Central US

ID	RESOURCE ID	COLLECTIONS
ToDoList	xiY+AA==	1

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.



Database: ToDoList
Collection: Items

```
SELECT * FROM c
```

REQUEST CHARGE 1.52941176470588 RUs
SHOWING RESULTS 1-2

```
[
  {
    "id": "be7fb49e-ba9e-4cd6-a832-76f25855909a",
    "name": "Deploy Template",
    "description": "Deploy DocumentDB and a Web application with an Azure Resource Manager template",
    "isComplete": true,
    "_self": "dbs/7XI5AA==/colls/7XI5AKQTbgE=/docs/7XI5AKQTbgEBAAAAAAAAAA==/",
    "_rid": "7XI5AKQTbgEBAAAAAAAAAA==",
    "_ts": 1423091699,
    "_etag": "\"0000c205-0000-0000-0000-54d2a7f30000\"",
    "_attachments": "attachments/"
  },
  {
    "id": "314449f2-df2c-4e1c-a50d-e5ea956d2e1b",
    "name": "Deploy another template",
    "description": "Deploy a different Azure Resource Manager template ",
    "isComplete": false,
    "_rid": "7XI5AKQTbgECAAAAAAAAAA==",
    "_ts": 1423091714,
    "_self": "dbs/7XI5AA==/colls/7XI5AKQTbgE=/docs/7XI5AKQTbgECAAAAAAAAAA==/",
    "_etag": "\"0000c305-0000-0000-0000-54d2a8020000\"",
    "_attachments": "attachments/"
  }
]
```

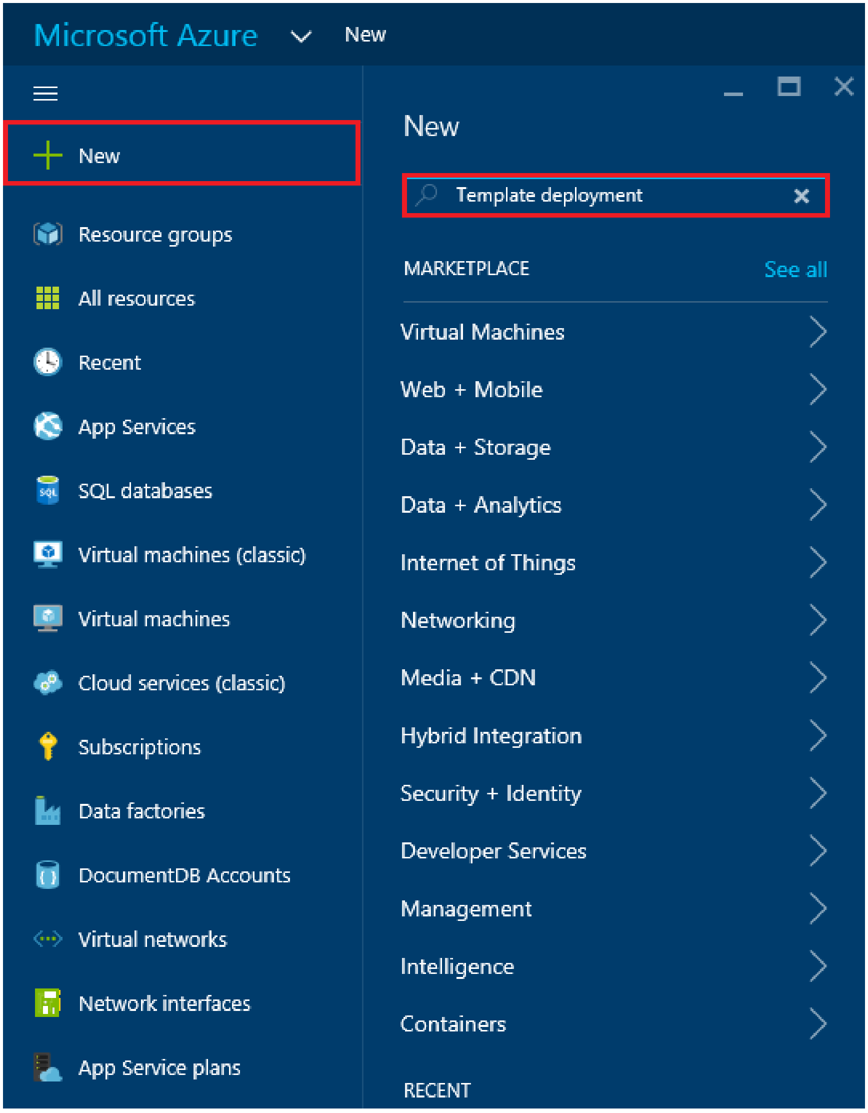
11. Feel free to explore the Azure Cosmos DB portal experience or modify the sample Todo application. When you're ready, let's deploy another template.

Step 3: Deploy the Document account and web app sample

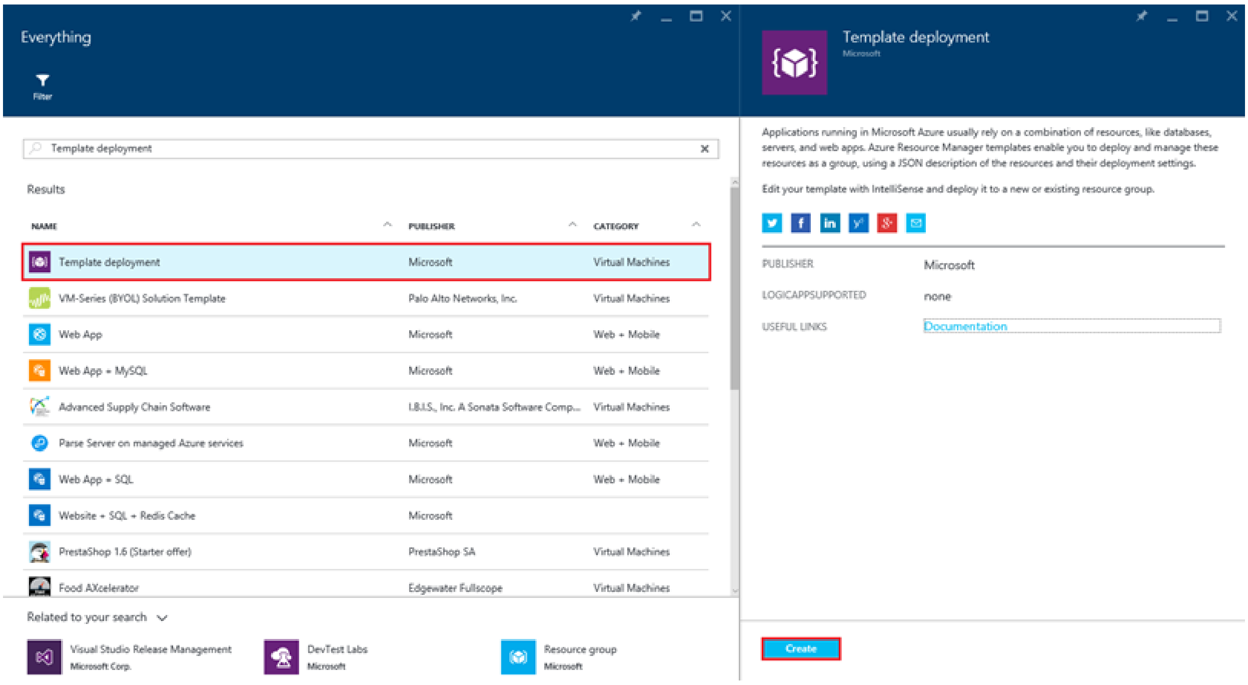
Now let's deploy our second template. This template is useful to show how you can inject Azure Cosmos DB connection information such as account endpoint and master key into a web app as application settings or as a custom connection string. For example, perhaps you have your own web application that you would like to deploy with an Azure Cosmos DB account and have the connection information automatically populated during deployment.

> TIP
>
> The template does not validate that the web app name and Azure Cosmos DB 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**



Results

NAME	PUBLISHER	CATEGORY
Template deployment	Microsoft	Virtual Machines
VM-Series (BYOL) Solution Template	Palo Alto Networks, Inc.	Virtual Machines
Web App	Microsoft	Web + Mobile
Web App + MySQL	Microsoft	Web + Mobile
Advanced Supply Chain Software	I.B.I.S., Inc. A Sonata Software Comp...	Virtual Machines
Parse Server on managed Azure services	Microsoft	Web + Mobile
Web App + SQL	Microsoft	Web + Mobile
Website + SQL + Redis Cache	Microsoft	
PrestaShop 1.6 (Starter offer)	PrestaShop SA	Virtual Machines
Food AXcelerator	Edgewater Fullscope	Virtual Machines

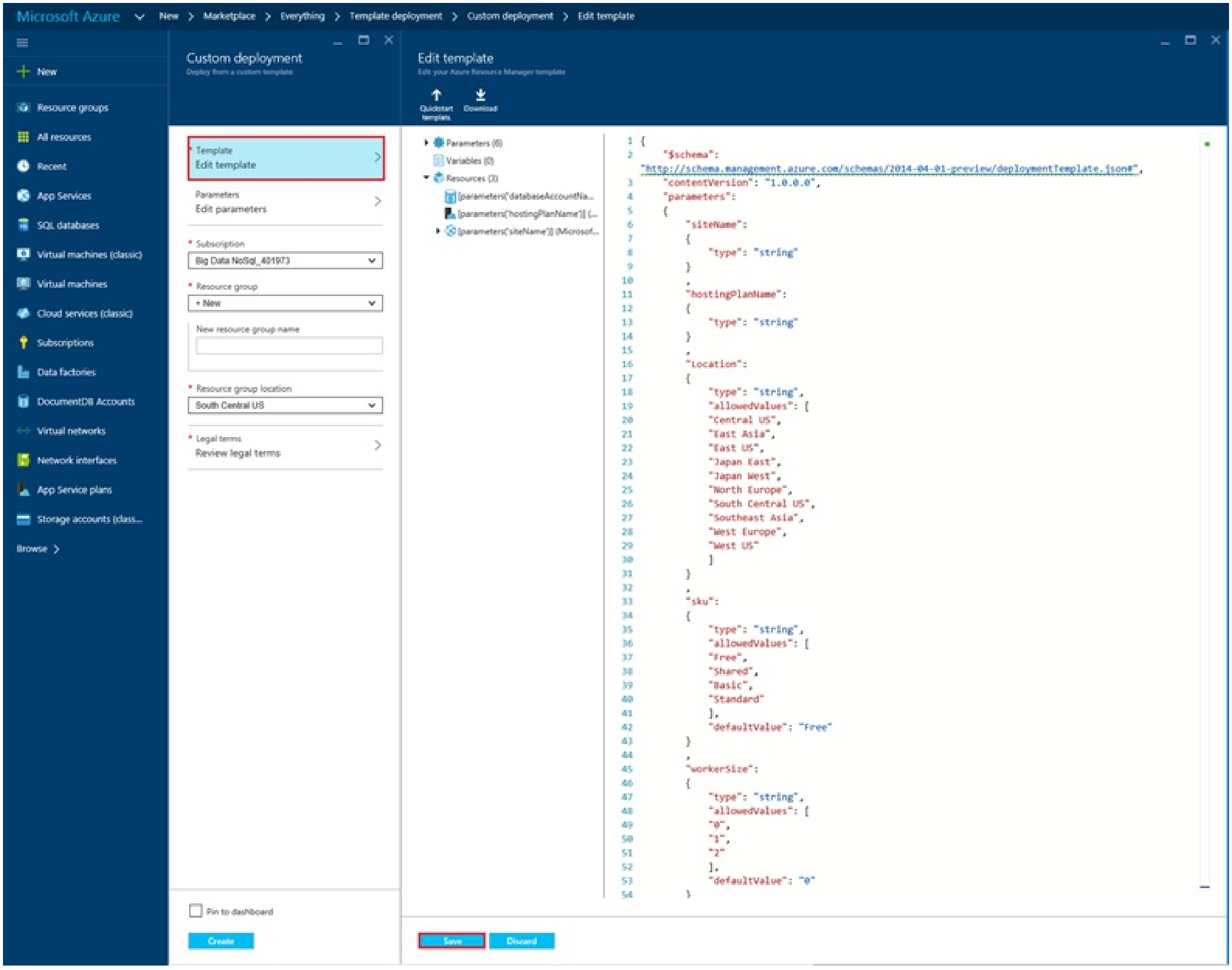
Template deployment
Microsoft

Applications running in Microsoft Azure usually rely on a combination of resources, like databases, servers, and web apps. Azure Resource Manager templates enable you to deploy and manage these resources as a group, using a JSON description of the resources and their deployment settings.

Edit your template with IntelliSense and deploy it to a new or existing resource group.

PUBLISHER	Microsoft
LOGICAPPSUPPORTED	none
USEFUL LINKS	Documentation

3. Click **Edit template**, paste the contents of the DocDBWebSite.json template file, and click **Save**.



* Subscription: Big Data NoSql_401973

* Resource group: + New

New resource group name

* Resource group location: South Central US

```
{
    "$schema":
"http://schema.management.azure.com/schemas/2014-04-01-preview/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters":
    {
        "siteName":
        {
            "type": "string"
        }
        ,
        "hostingPlanName":
        {
            "type": "string"
        }
        ,
        "location":
        {
            "type": "string",
            "allowedValues": [
            "Central US",
            "East Asia",
            "East US",
            "Japan East",
            "Japan West",
            "North Europe",
            "South Central US",
            "Southeast Asia",
            "West Europe",
            "West US"
            ]
        }
        ,
        "sku":
        {
            "type": "string",
            "allowedValues": [
            "Free",
            "Shared",
            "Basic",
            "Standard"
            ],
            "defaultValue": "Free"
        }
        ,
        "workerSize":
        {
            "type": "string",
            "allowedValues": [
            "0",
            "1",
            "2"
            ],
            "defaultValue": "0"
        }
```

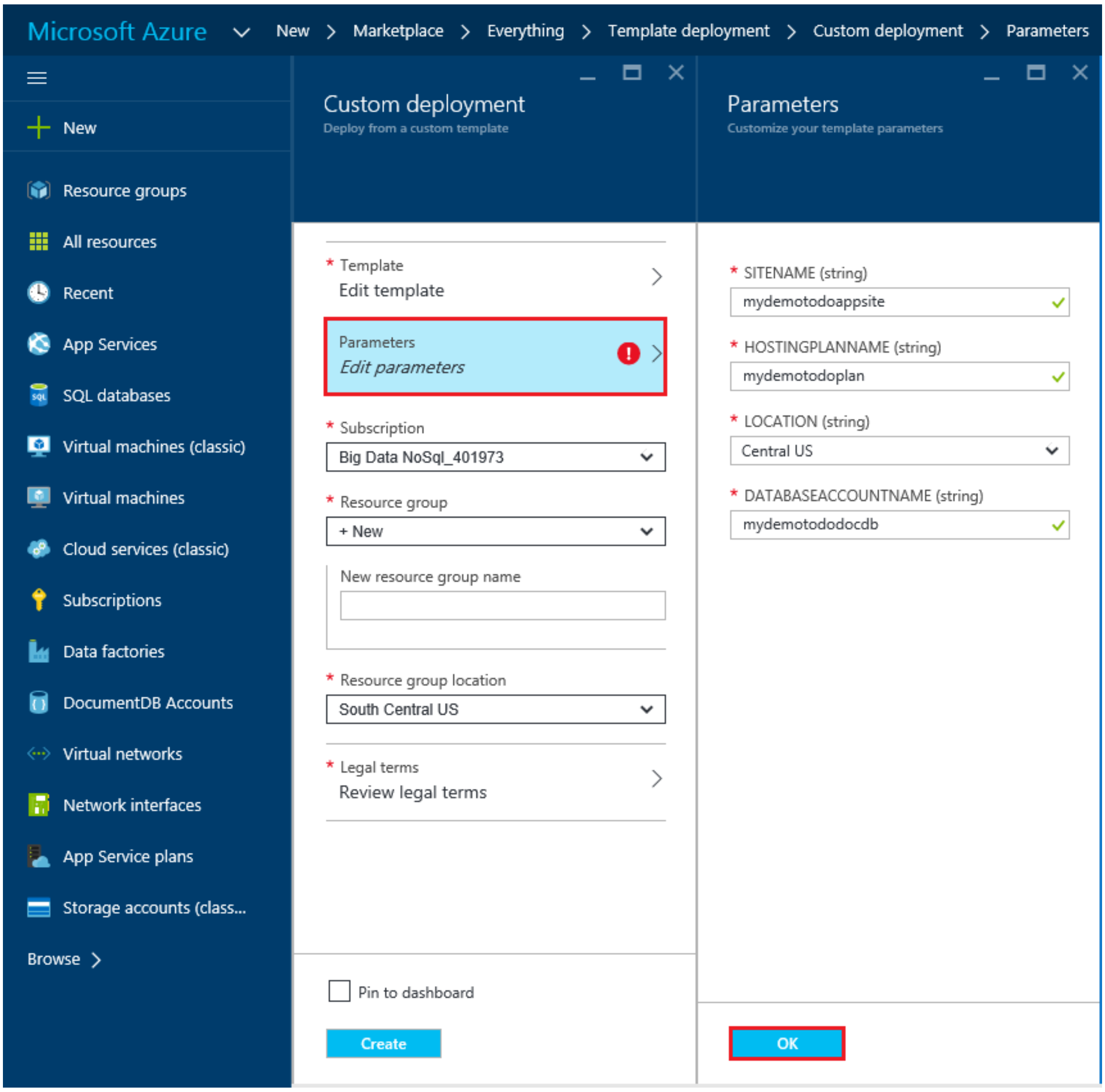
4. Click **Edit parameters**, provide values for each of the mandatory parameters, and click **OK**. The parameters are as follows:

   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 Azure Cosmos DB and web app resources.

   d. DATABASEACCOUNTNAME: Specifies the name of the Azure Cosmos DB account to create.



* 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

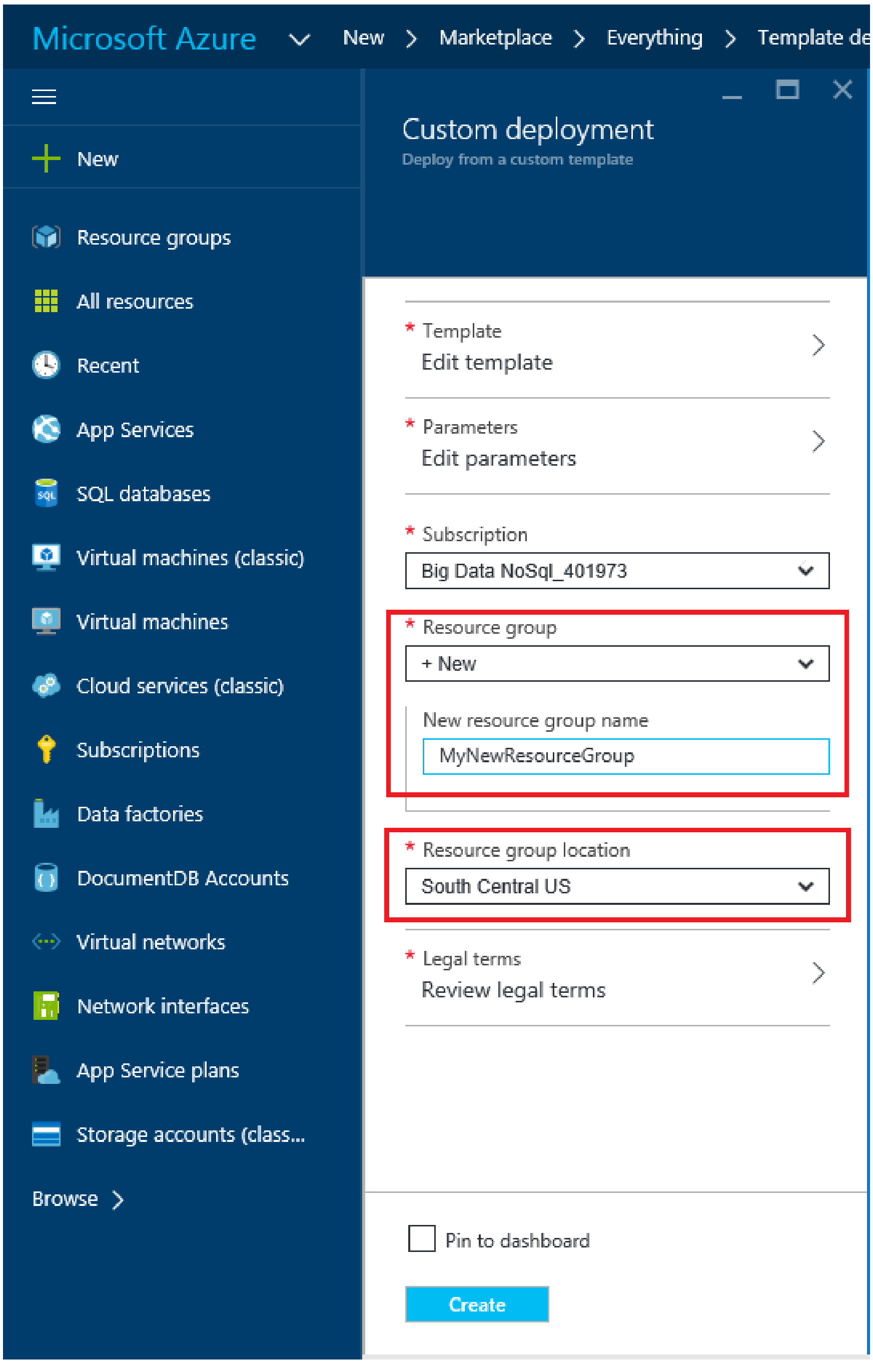
* SITENAME (string): mydemotodoappsite ✓

* HOSTINGPLANNAME (string): mydemotodoplan ✓

* LOCATION (string): Central US

* DATABASEACCOUNTNAME (string): mydemotododocdb ✓

5. Choose an existing Resource group or provide a name to make a new resource group, and choose a location for the resource group.



* Template
Edit template

* Parameters
Edit parameters

* Subscription
Big Data NoSql_401973

* Resource group
+ New

New resource group name
MyNewResourceGroup

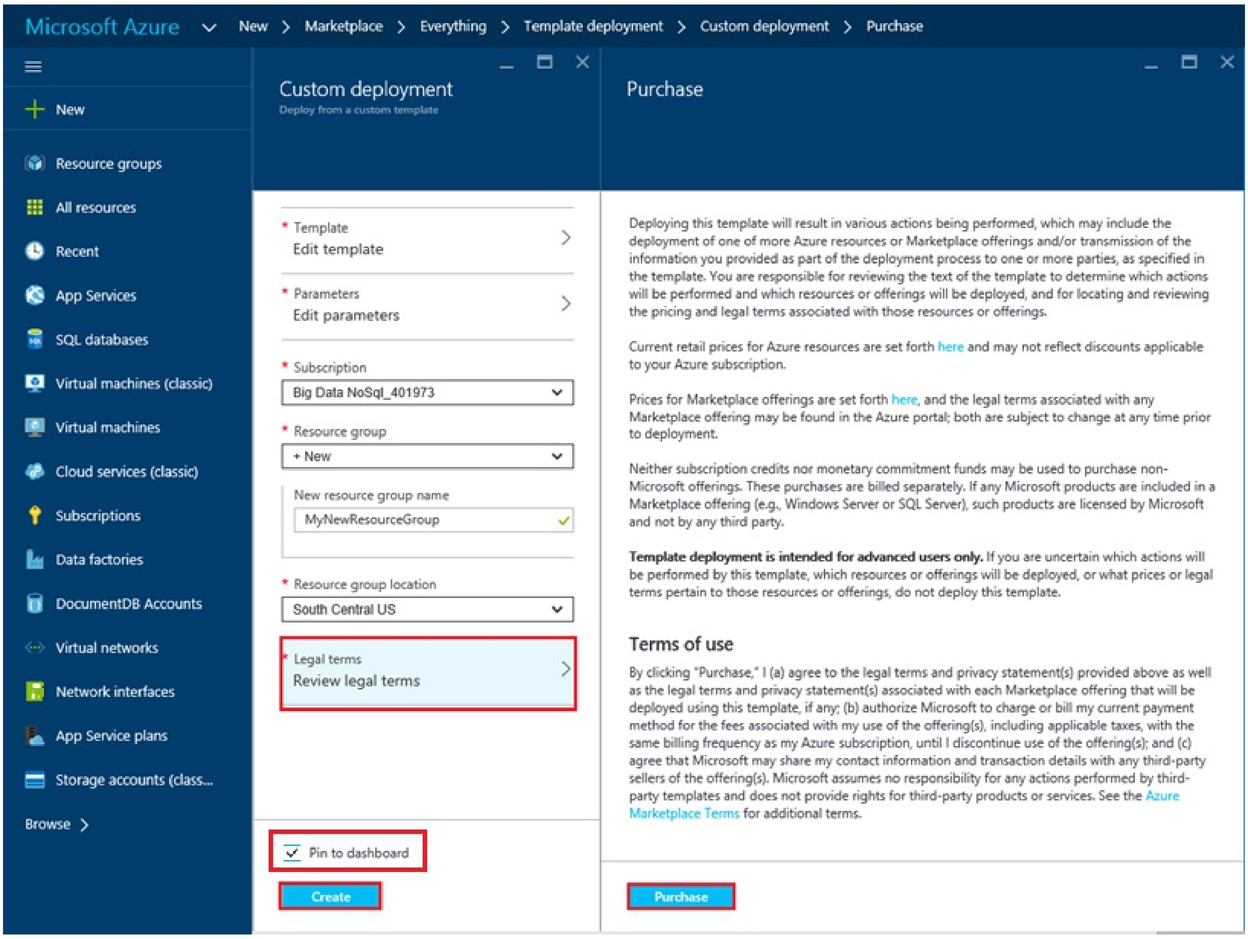
* Resource group location
South Central US

* Legal terms
Review legal terms

□ Pin to dashboard

Create

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.



* 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

Deploying this template will result in various actions being performed, which may include the deployment of one of more Azure resources or Marketplace offerings and/or transmission of the information you provided as part of the deployment process to one or more parties, as specified in the template. You are responsible for reviewing the text of the template to determine which actions will be performed and which resources or offerings will be deployed, and for locating and reviewing the pricing and legal terms associated with those resources or offerings.

Current retail prices for Azure resources are set forth here and may not reflect discounts applicable to your Azure subscription.

Prices for Marketplace offerings are set forth here, and the legal terms associated with any Marketplace offering may be found in the Azure portal; both are subject to change at any time prior to deployment.

Neither subscription credits nor monetary commitment funds may be used to purchase non-Microsoft offerings. These purchases are billed separately. If any Microsoft products are included in a Marketplace offering (e.g., Windows Server or SQL Server), such products are licensed by Microsoft and not by any third party.

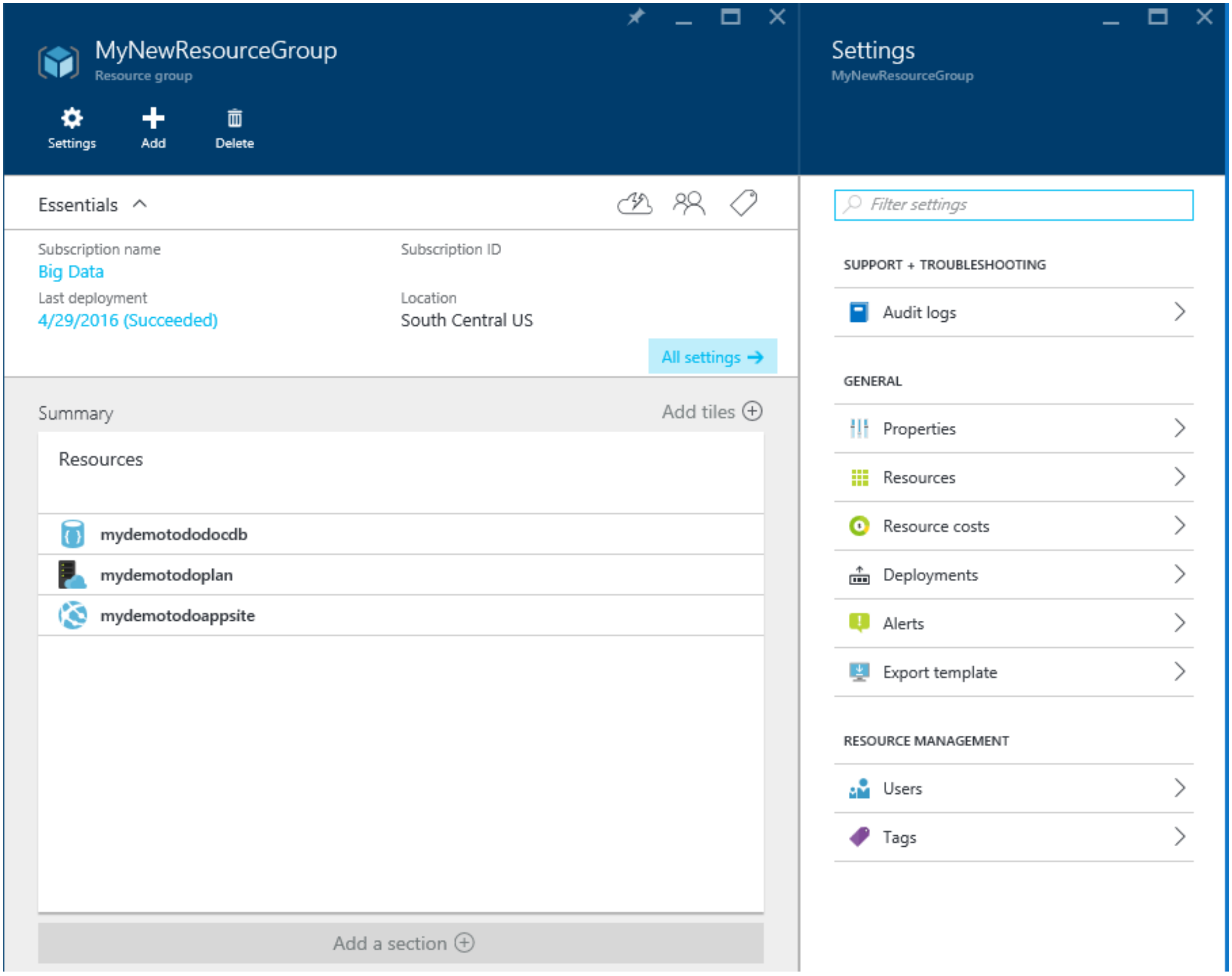
Template deployment is intended for advanced users only. If you are uncertain which actions will be performed by this template, which resources or offerings will be deployed, or what prices or legal terms pertain to those resources or offerings, do not deploy this template.

Terms of use

By clicking "Purchase," I (a) agree to the legal terms and privacy statement(s) provided above as well as the legal terms and privacy statement(s) associated with each Marketplace offering that will be deployed using this template, if any; (b) authorize Microsoft to charge or bill my current payment method for the fees associated with my use of the offering(s), including applicable taxes, with the same billing frequency as my Azure subscription, until I discontinue use of the offering(s); and (c) agree that Microsoft may share my contact information and transaction details with any third-party sellers of the offering(s). Microsoft assumes no responsibility for any actions performed by third-party templates and does not provide rights for third-party products or services. See the Azure Marketplace Terms for additional terms.

Purchase

7. When the deployment finishes, the Resource group blade will open.



Subscription name
Big Data

Subscription ID

Last deployment
4/29/2016 (Succeeded)

Location
South Central US

All settings →

Summary

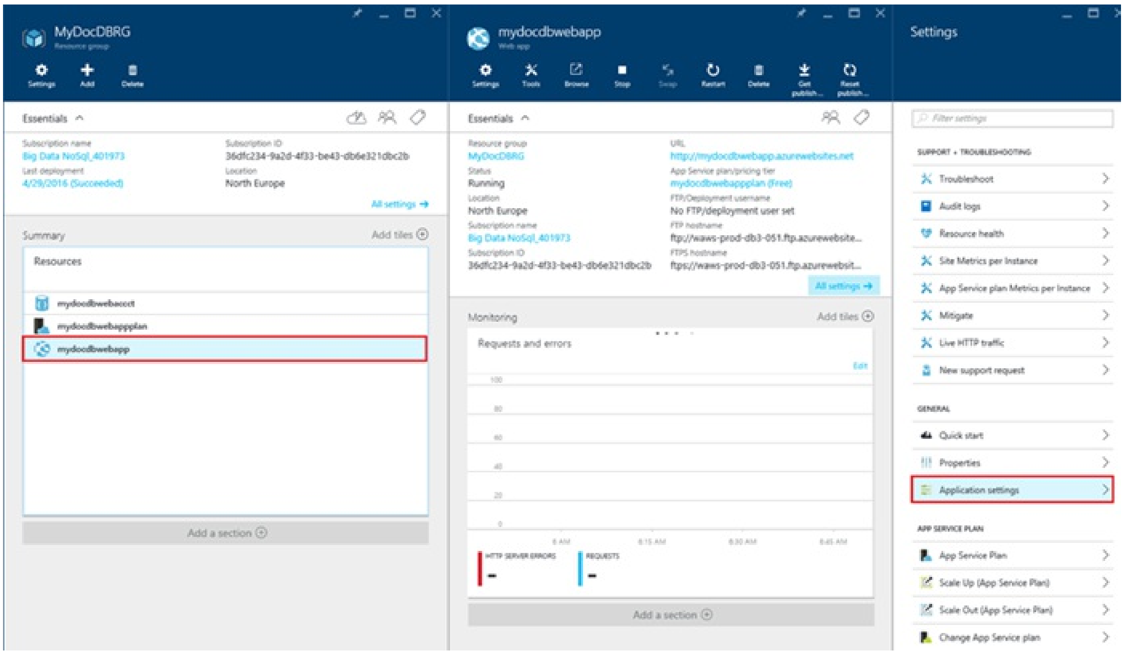
Add tiles ⊕

Resources

mydemotodododocdb
mydemotodoplan
mydemotodoappsite

Add a section ⊕

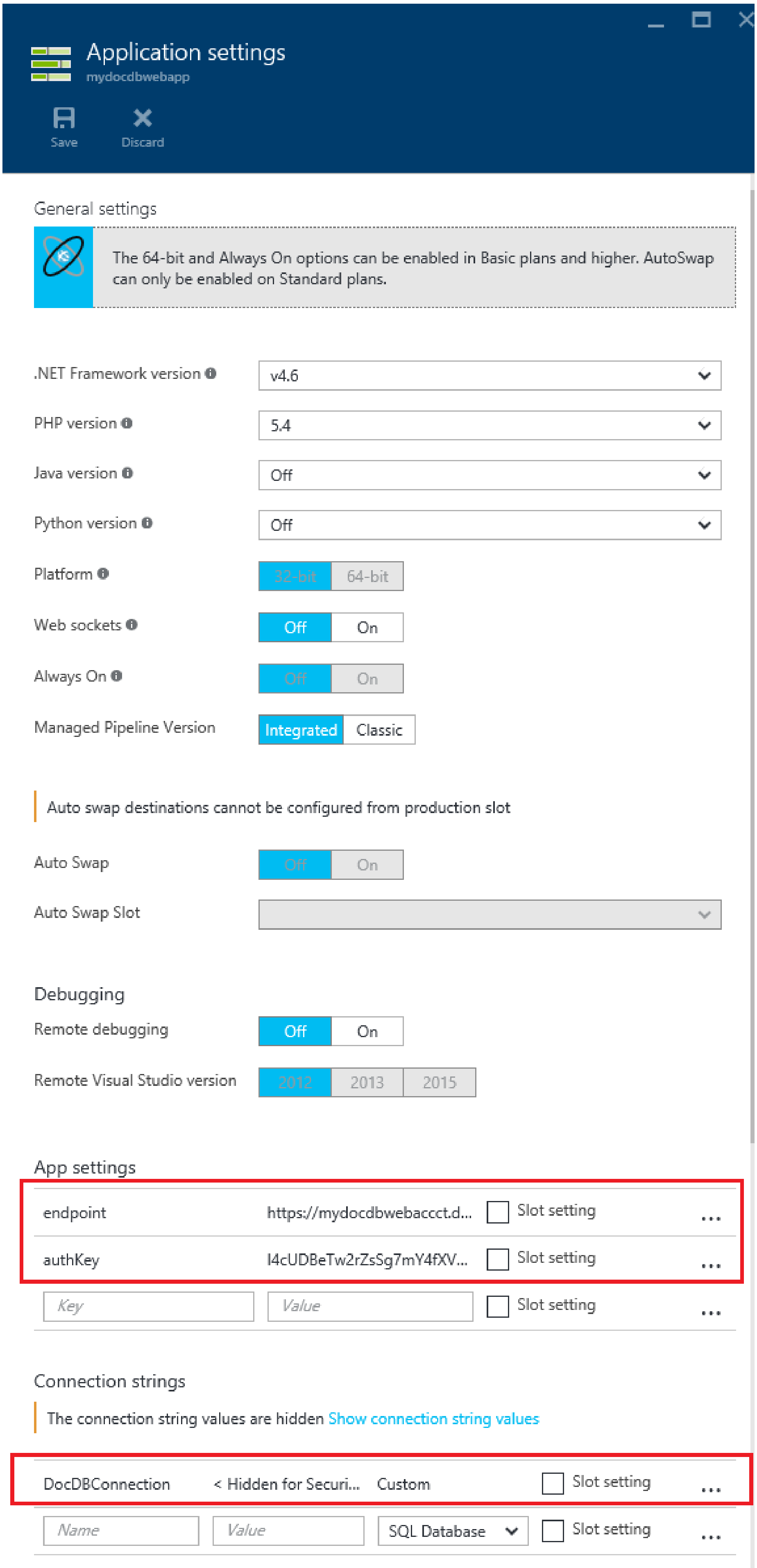
8. Click the Web App resource in the Resources list and then click **Application settings**



Subscription name: Big Data NoSql_401973
Subscription ID: 36dfc234-9a2d-4f33-be43-db6e321dbc2b
Last deployment: 4/29/2016 (Succeeded)
Location: North Europe

Resource group: MyDocDBRG
URL: http://mydocdbwebapp.azurewebsites.net
Status: Running
App Service plan/pricing tier: mydocdbwebappplan (Free)
Location: North Europe
FTP/Deployment username: No FTP/deployment user set
Subscription name: Big Data NoSql_401973
FTP hostname: ftp://waws-prod-db3-051.ftp.azurewebsite...
Subscription ID: 36dfc234-9a2d-4f33-be43-db6e321dbc2b
FTPS hostname: ftps://waws-prod-db3-051.ftp.azurewebsit...

9. Note how there are application settings present for the Azure Cosmos DB endpoint and each of the Azure Cosmos DB master keys.



Application settings
mydocdbwebapp

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-bit] 64-bit

Web sockets: [✓ Off] On

Always On: [✓ Off] On

Managed Pipeline Version: [✓ Integrated] Classic

Auto swap destinations cannot be configured from production slot

Auto Swap: [✓ Off] On

Auto Swap Slot:

Debugging

Remote debugging: [✓ Off] On

Remote Visual Studio version: [✓ 2012] 2013 2015

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	...

10. Feel free to continue exploring the Azure Portal, or follow one of our Azure Cosmos DB samples to create your own Azure Cosmos DB application.

Next steps

Congratulations! You've deployed Azure Cosmos DB, App Service web app and a sample web application using Azure Resource Manager templates.

- To learn more about Azure Cosmos DB, 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.

Scenario: Exception handling and error logging for logic apps

5/24/2017 • 7 min to read • Edit Online

This scenario describes how you can extend a logic app to better support exception handling. We've used a real-life use case to answer the question: "Does Azure Logic Apps support exception and error handling?"

NOTE

The current Azure Logic Apps schema provides a standard template for action responses. This template includes both internal validation and error responses returned from an API app.

Scenario and use case overview

Here's the story as the use case for this scenario:

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

TIP

For a high-level video about this project, see Integration User Group.

How we solved the problem

We chose Azure Cosmos DB As a repository for the log and error records (Cosmos DB refers to records as documents). Because Azure 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. Cosmos DB has a property called Time to Live (TTL), which allowed us to set a **Time to Live** value for each record or collection. This capability eliminated the need to manually delete records in Cosmos DB.

IMPORTANT

To complete this tutorial, you need to create a Cosmos DB database and two collections (Logging and Errors).

Create the logic app

The first step is to create the logic app and open the app in Logic App 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 log the record coming out of Dynamics CRM Online, let's start at the top. We must use a **Request** trigger because the parent logic app triggers this child.

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

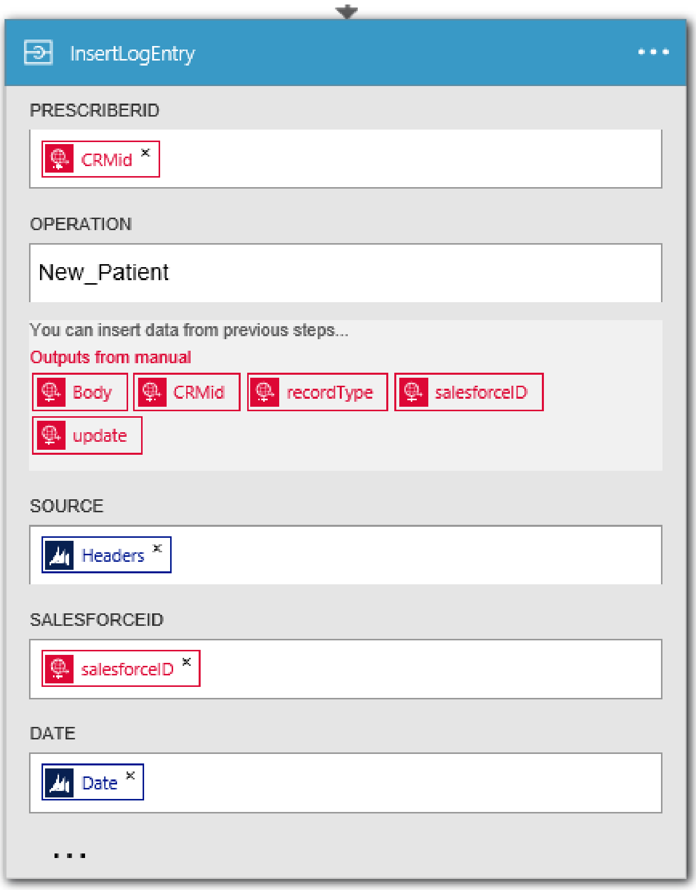
We must log the source (request) of the patient record from the Dynamics CRM Online portal.

1. We must get a new appointment record from Dynamics CRM Online.

   The trigger coming from CRM provides us with the **CRM PatentId**, **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 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 here in Logic App Designer.

   Insert log entry



InsertLogEntry

PRESCRIBERID

CRMid

OPERATION

New_Patient

You can insert data from previous steps...

Outputs from manual

SOURCE

Headers

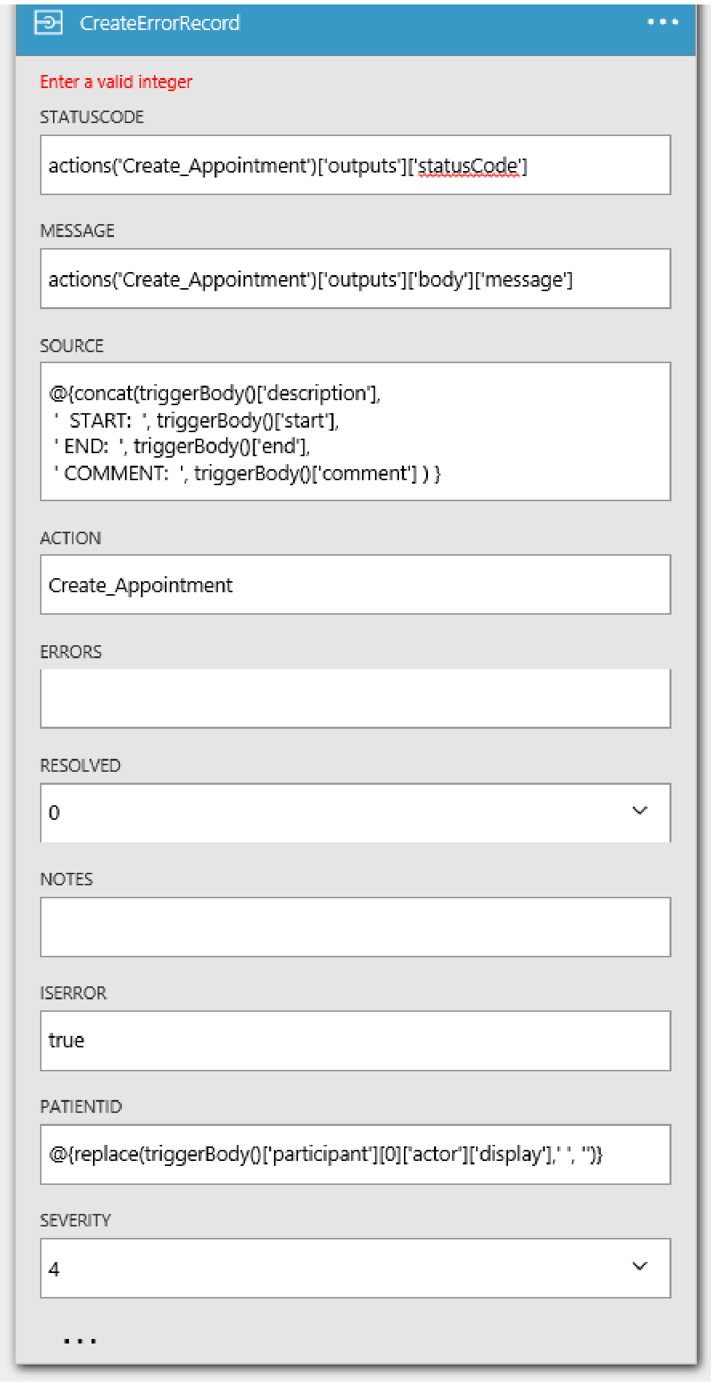
SALESFORCEID

salesforceID

DATE

Date

Insert error entry



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

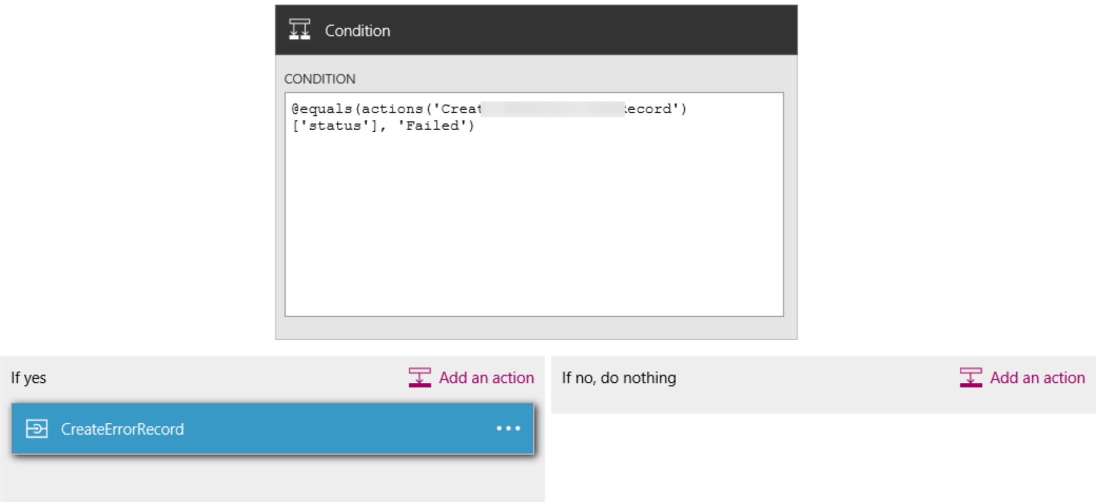
PATIENTID

@{replace(triggerBody()['participant'][0]['actor']['display'],' ', '')}

SEVERITY

4

Check for create record failure



CONDITION

```
@equals(actions('Creat[redacted]ecord')
['status'], 'Failed')
```

Logic app source code

NOTE

The following examples are samples only. Because this tutorial is based on an implementation now 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

Here 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_NewPatientecord": ["Succeeded"]
    }
}
```

Log request

Here 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": "{/"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/":/"Micros
oft-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

Here 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": "XngRAOT6IQEHAAAAAAAAAA==",
      "_self": "dbs/XngRAA==/colls/XngRAOT6IQE=/docs/XngRAOT6IQEHAAAAAAAAAA==/",
      "_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/":/"Micros
oft-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 app code sample shows how you can implement error handling.

Create error record

Here is the logic app 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 Cosmos DB--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/":/"/",/"FirstName/":/"A/",/"Gender_vod__c/":/"/",/"IMS_ID__c/":/"/"
,/"LastName/":/"BAILEY/",/"MasterID_mp__c/":/"/",/"C_ID_MED__c/":/"851588/",/"Middle_vod__c/":/"/",/"NPI_vod__c/":/"/",/"PDRP_MED
__c/":false,/"PersonDoNotCall/":false,/"PersonEmail/":/"/",/"PersonHasOptedOutOfEmail/":false,/"PersonHasOptedOutOfFax/":false,/"PersonMo
bilePhone/":/"/",/"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/",/"ONY_ID__c/":/"/",/"Specialty_1_vod__c/":/"/",/"Suffix_vod__c/":/"/",/"Website/":/"/"}",
      "statusCode": "400"
   }
}
```

Insert error into Cosmos DB--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": "sQx2APhVzAA8AAAAAAAAAA==",
    "_self": "dbs/sQx2AA==/colls/sQx2APhVzAA=/docs/sQx2APhVzAA8AAAAAAAAAA==/",
    "_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/",/"MterI
D_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/",/"XXXXXXX/":/"/",/"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": []
  }
}
```

Return the response back to parent logic app

After you get the response, you can pass the response back to the parent logic app.

Return success response to 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 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"
}
```

Cosmos DB repository and portal

Our solution added capabilities with Cosmos DB.

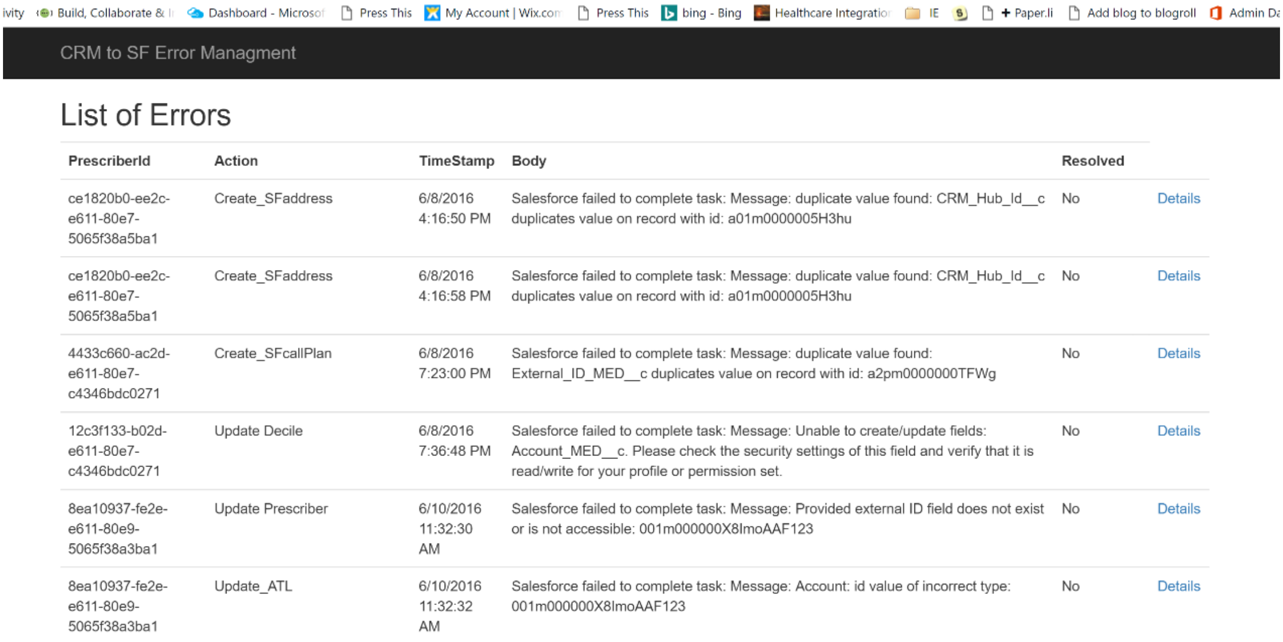
Error management portal

To view the errors, you can create an MVC web app to display the error records from Cosmos DB. The **List**, **Details**, **Edit**, and **Delete** operations are included in the current version.

> NOTE
>
> Edit operation: Cosmos DB replaces the entire document. The records shown in the **List** and **Detail** views are samples only. They are not actual patient appointment records.

Here are examples of our MVC app details created with the previously described approach.

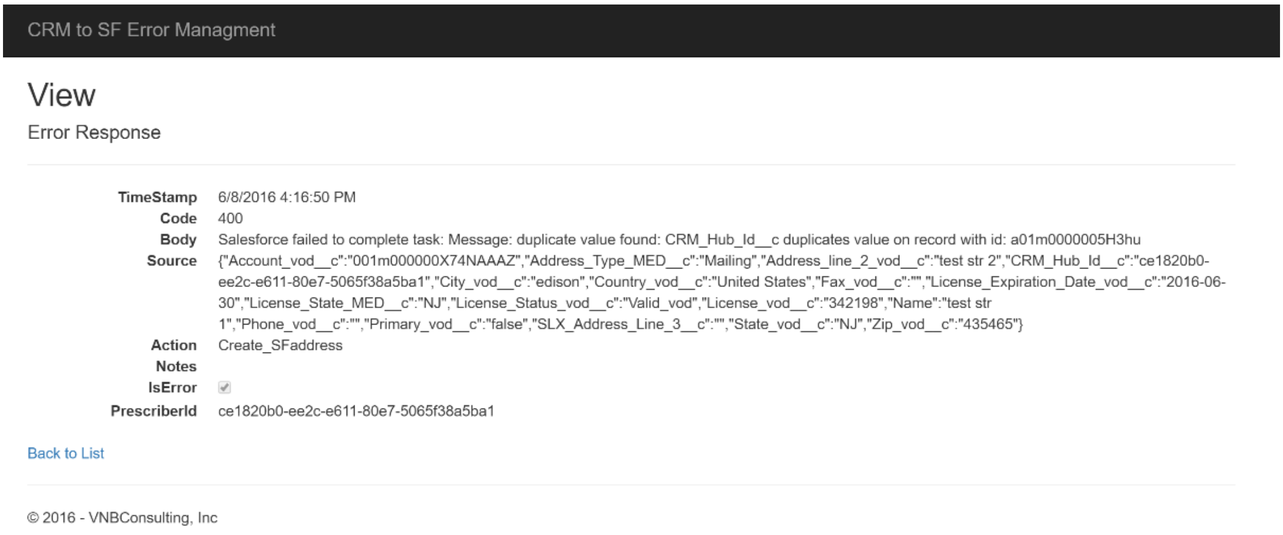
Error management list



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: 001m000000X8ImoAAF123	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: 001m000000X8ImoAAF123	No	Details

Error management detail view



View

Error Response

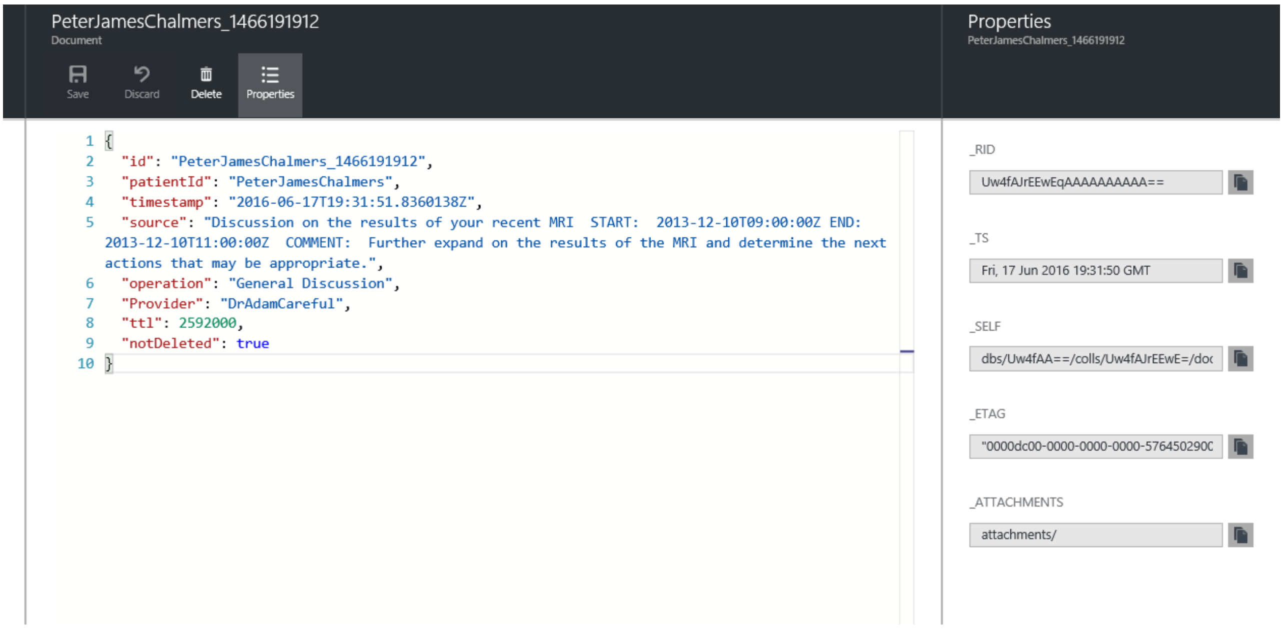
TimeStamp	6/8/2016 4:16:50 PM
Code	400
Body	Salesforce failed to complete task: Message: duplicate value found: CRM_Hub_Id__c duplicates value on record with id: a01m0000005H3hu
Source	{"Account_vod__c":"001m000000X74NAAAZ","Address_Type_MED__c":"Mailing","Address_line_2_vod__c":"test str 2","CRM_Hub_Id__c":"ce1820b0-ee2c-e611-80e7-5065f38a5ba1","City_vod__c":"edison","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

© 2016 - VNBConsulting, Inc

Log management portal

To view the logs, we also created an MVC web app. Here are examples of our MVC app details created with the previously described approach.

Sample log detail view



```
{
  "id": "PeterJamesChalmers_1466191912",
  "patientId": "PeterJamesChalmers",
  "timestamp": "2016-06-17T19:31:51.8360138Z",
  "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.",
  "operation": "General Discussion",
  "Provider": "DrAdamCareful",
  "ttl": 2592000,
  "notDeleted": true
}
```

_RID: Uw4fAJrEEwEqAAAAAAAAAA==

_TS: Fri, 17 Jun 2016 19:31:50 GMT

_SELF: dbs/Uw4fAA==/colls/Uw4fAJrEEwE=/doc

_ETAG: "0000dc00-0000-0000-0000-57645029OC

_ATTACHMENTS: attachments/

API app details

Logic Apps exception management API

Our open-source Azure Logic Apps exception management API app provides functionality as described here - 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, allowing operations to resolve 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 the value 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 checks 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 app examples and scenarios
- Learn about monitoring logic apps
- Create automated deployment templates for logic apps

Azure Functions Cosmos DB bindings

5/10/2017 • 5 min to read • Edit Online

This article explains how to configure and code Azure Cosmos DB bindings in Azure Functions. Azure Functions supports input and output bindings for Cosmos DB.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first function
- Azure Functions developer reference
- C#, F#, or Node developer reference
- Azure Functions triggers and bindings concepts

For more information on Cosmos DB, see Introduction to Cosmos DB and Build a Cosmos DB console application.

DocumentDB API input binding

The DocumentDB API input binding retrieves a Cosmos DB 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 API input binding has the following properties in *function.json*:

- `name` : Identifier name used in function code for the document
- `type` : must be set to "documentdb"
- `databaseName` : The database containing the document
- `collectionName` : The collection containing the document
- `id` : The Id of the document to retrieve. This property supports bindings parameters; see Bind to custom input properties in a binding expression in the article Azure Functions triggers and bindings concepts.
- `sqlQuery` : A Cosmos DB SQL query used for retrieving multiple documents. The query supports runtime bindings. For example: `SELECT * FROM c where c.departmentId = {departmentId}`
- `connection` : The name of the app setting containing your Cosmos DB connection string
- `direction` : must be set to `"in"`.

The properties `id` and `sqlQuery` cannot both be specified. If neither `id` nor `sqlQuery` is set, the entire collection is retrieved.

Using a DocumentDB API input binding

- In C# and F# functions, when the function exits successfully, any changes made to the input document via named input parameters are automatically persisted.
- In JavaScript functions, updates are not made automatically upon function exit. Instead, use `context.bindings.<documentName>In` and `context.bindings.<documentName>Out` to make updates. See the JavaScript sample.

Input sample for single document

Suppose you have the following DocumentDB API input binding in the `bindings` array of function.json:

```
{
  "name": "inputDocument",
  "type": "documentDB",
  "databaseName": "MyDatabase",
  "collectionName": "MyCollection",
  "id" : "{queueTrigger}",
  "connection": "MyAccount_COSMOSDB",
  "direction": "in"
}
```

See the language-specific sample that uses this input binding to update the document's text value.

- C#
- F#
- JavaScript

Input sample in C#

```
// Change input document contents using DocumentDB API input binding
public static void Run(string myQueueItem, dynamic inputDocument)
{
  inputDocument.text = "This has changed.";
}
```

Input sample in F#

```
(* Change input document contents using DocumentDB API input binding *)
open FSharp.Interop.Dynamic
let Run(myQueueItem: string, inputDocument: obj) =
  inputDocument?text <- "This has changed."
```

This sample requires 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 JavaScript

```
// Change input document contents using DocumentDB API input binding, using context.bindings.inputDocumentOut
module.exports = function (context) {
  context.bindings.inputDocumentOut = context.bindings.inputDocumentIn;
  context.bindings.inputDocumentOut.text = "This was updated!";
  context.done();
};
```

Input sample with multiple documents

Suppose that you wish to retrieve multiple documents specified by a SQL query, using a queue trigger to customize the query parameters.

In this example, the queue trigger provides a parameter `departmentId`.A queue message of `{ "departmentId" : "Finance" }` would return all records for the finance department. Use the following in *function.json*:

```
{
  "name": "documents",
  "type": "documentdb",
  "direction": "in",
  "databaseName": "MyDb",
  "collectionName": "MyCollection",
  "sqlQuery": "SELECT * from c where c.departmentId = {departmentId}"
  "connection": "CosmosDBConnection"
}
```

Input sample with multiple documents in C#

```
public static void Run(QueuePayload myQueueItem, IEnumerable<dynamic> documents)
{
  foreach (var doc in documents)
  {
    // operate on each document
  }
}

public class QueuePayload
{
  public string departmentId { get; set; }
}
```

Input sample with multiple documents in JavaScript

```
module.exports = function (context, input) {
  var documents = context.bindings.documents;
  for (var i = 0; i < documents.length; i++) {
    var document = documents[i];
    // operate on each document
  }
  context.done();
};
```

DocumentDB API output binding

The DocumentDB API output binding lets you write a new document to an Azure Cosmos DB database. It has the following properties in *function.json*:

- `name` : Identifier used in function code for the new document
- `type` : must be set to `"documentdb"`
- `databaseName` : The database containing the collection where the new document will be created.
- `collectionName` : The collection where the new document will be created.
- `createIfNotExists` : A boolean value to indicate whether the collection will be created if it does not exist. The default is *false*. The reason for this is new collections are created with reserved throughput, which has pricing implications. For more details, please visit the pricing page.
- `connection` : The name of the app setting containing your Cosmos DB connection string
- `direction` : must be set to `"out"`

Using a DocumentDB API output binding

This section shows you how to use your DocumentDB API 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.

> NOTE
>
> When you specify the ID of an existing document, it gets overwritten by the new output document.

To output multiple documents, you can also bind to `ICollector<T>` or `IAsyncCollector<T>` where `T` is one of the supported types.

DocumentDB API output binding sample

Suppose you have the following DocumentDB API output binding in the `bindings` array of function.json:

```
{
 "name": "employeeDocument",
 "type": "documentDB",
 "databaseName": "MyDatabase",
 "collectionName": "MyCollection",
 "createIfNotExists": true,
 "connection": "MyAccount_COSMOSDB",
 "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 Cosmos DB 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#
- JavaScript

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 }
```

This sample requires 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 JavaScript

```
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 an Apache Hive, Pig, or Hadoop job using Azure Cosmos DB and HDInsight

6/9/2017 • 15 min to read • Edit Online

This tutorial shows you how to run Apache Hive, Apache Pig, and Apache Hadoop MapReduce jobs on Azure HDInsight with Cosmos DB's Hadoop connector. Cosmos DB's Hadoop connector allows Cosmos DB to act as both a source and sink for Hive, Pig, and MapReduce jobs. This tutorial will use Cosmos DB 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 Cosmos DB using a Hive, Pig, or MapReduce job?
- How do I store data in Cosmos DB using a Hive, Pig, or MapReduce job?

We recommend getting started by watching the following video, where we run through a Hive job using Cosmos DB and HDInsight.

Then, return to this article, where you'll receive the full details on how you can run analytics jobs on your Cosmos DB 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 Cosmos DB and have a Cosmos DB account. If you are new to Cosmos DB or you do not have a Cosmos DB 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 Cosmos DB account, a database, and a collection with documents inside. For more information, see Getting Started with Cosmos DB. Import sample data into your Cosmos DB account with the Cosmos DB import tool.
- Throughput. Reads and writes from HDInsight will be counted towards your allotted request units for your collections.
- Capacity for an additional stored procedure within each output collection. The stored procedures are used for transferring resulting documents.
- Capacity for the resulting documents from the Hive, Pig, or MapReduce jobs.
- [*Optional*] Capacity for an additional collection.

> 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.

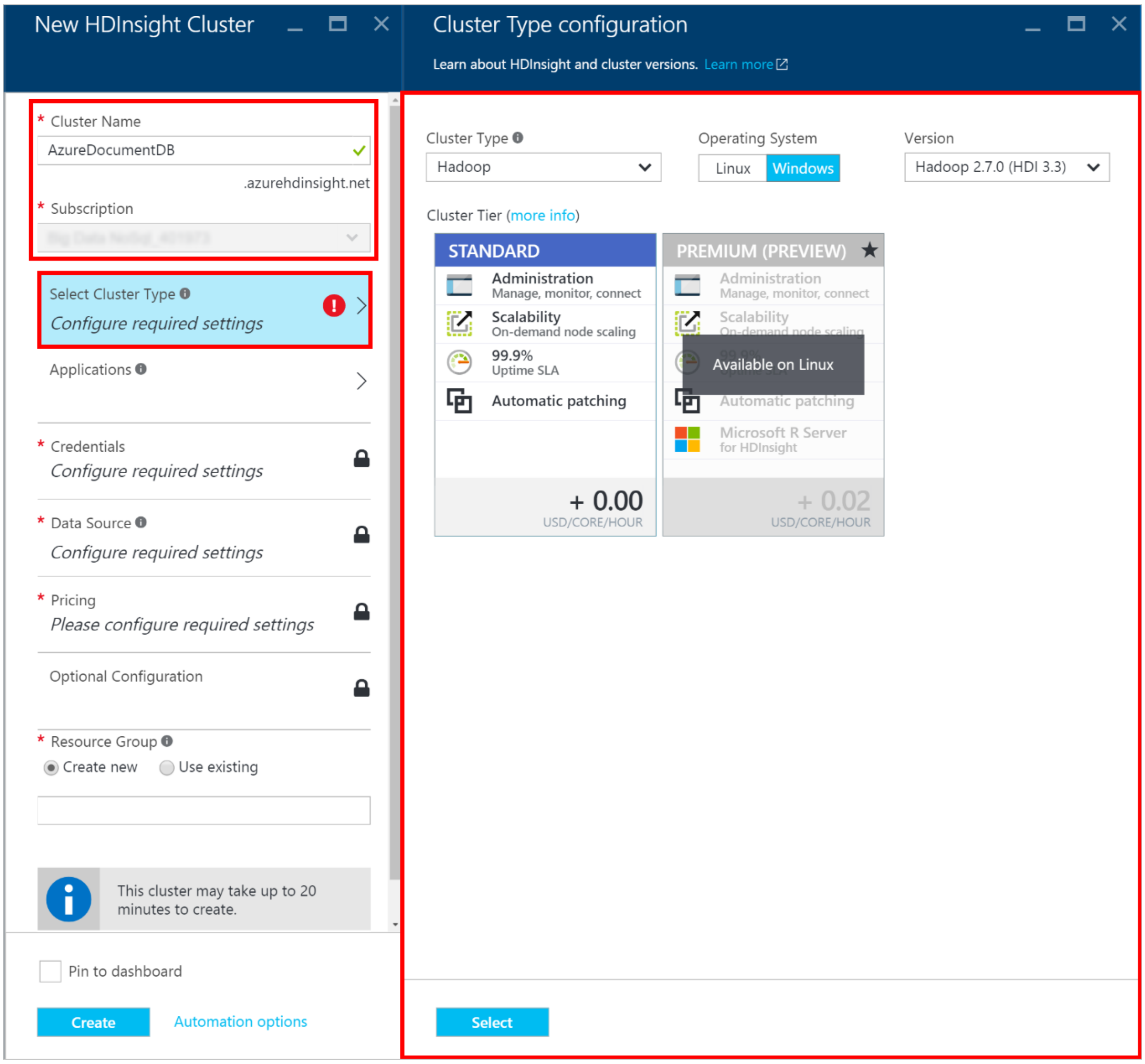
Cluster name	Name the cluster. DNS name must start and end with an alpha numeric character, and may contain dashes. The field must be a string between 3 and 63 characters.
Subscription Name	If you have more than one Azure Subscription, select the subscription that will host your HDInsight cluster.

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**.



New HDInsight Cluster

* Cluster Name
AzureDocumentDB
.azurehdinsight.net

* Subscription
[redacted]

Select Cluster Type
Configure required settings

Applications

* Credentials
Configure required settings

* Data Source
Configure required settings

* Pricing
Please configure required settings

Optional Configuration

* Resource Group
● Create new ○ Use existing

This cluster may take up to 20 minutes to create.

□ Pin to dashboard

Create Automation options

Cluster Type configuration

Learn about HDInsight and cluster versions. Learn more

Cluster Type: Hadoop
Operating System: Linux [✓ Windows]
Version: Hadoop 2.7.0 (HDI 3.3)

Cluster Tier (more info)

STANDARD	PREMIUM (PREVIEW)
Administration Manage, monitor, connect	Administration Manage, monitor, connect
Scalability On-demand node scaling	Scalability On-demand node scaling
99.9% Uptime SLA	Available on Linux
Automatic patching	Automatic patching
	Microsoft R Server for HDInsight
+ 0.00 USD/CORE/HOUR	+ 0.02 USD/CORE/HOUR

Select

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.

7. 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.
8. On the same blade, specify a **Default Container** and a **Location**. And, click **SELECT**.

> NOTE
>
> Select a location close to your Cosmos DB account region for better performance

9. 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.
10. 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**.

11. Create either a new **Resource Group** or use an existing Resource Group under your Azure Subscription.
12. Now, check **Pin to dashboard** to track its deployment and click **Create**!

Step 2: Install and configure Azure PowerShell

1. 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**.

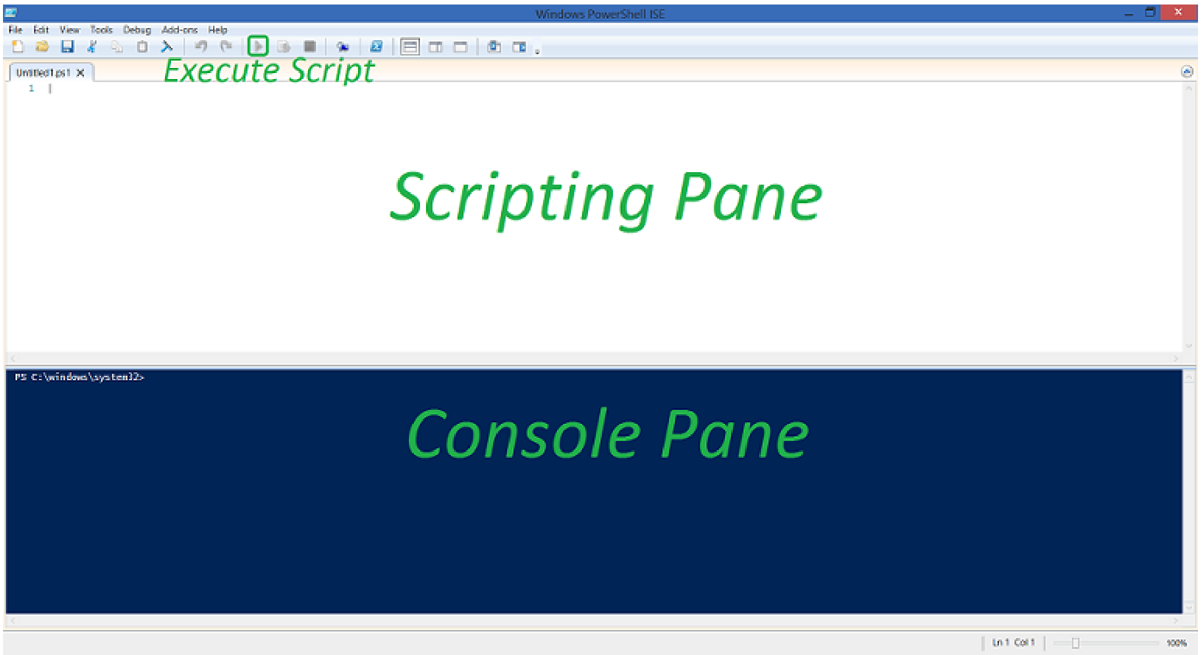
2. Open the Azure PowerShell Integrated Scripting Environment:
   - On a computer running Windows 8 or Windows Server 2012 or higher, you can use the built-in Search. From the Start screen, type **powershell ise** and click **Enter**.
   - On a computer running a version earlier than Windows 8 or Windows 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 Cosmos DB 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 t 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' ); "
```

3. 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>' ); "
```

4. 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)); "
```

5. 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.

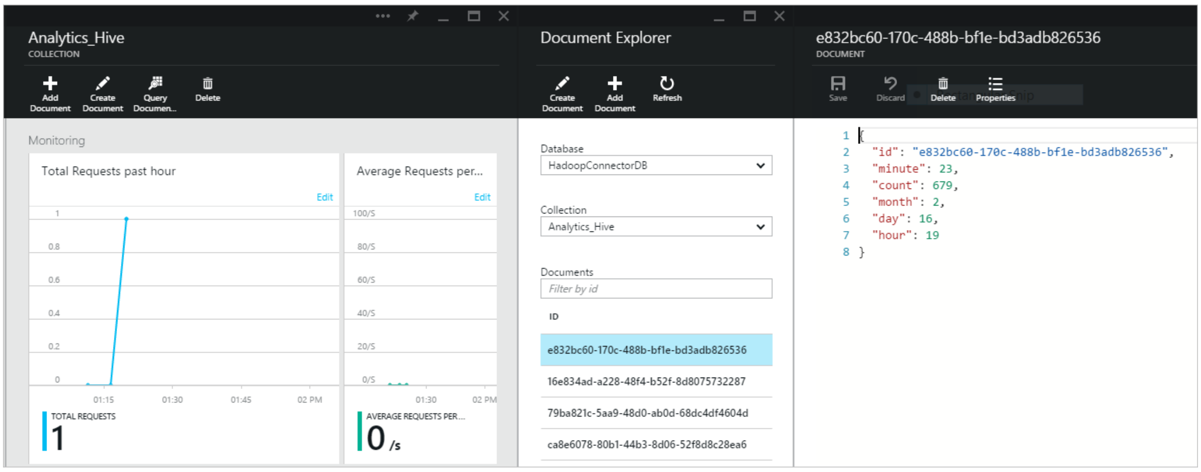
```
# Wait for the Hive job to complete.
Wait-AzureHDInsightJob -Job $hiveJob -WaitTimeoutInSeconds 3600
```

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.



Analytics_Hive
COLLECTION

Document Explorer

Database: HadoopConnectorDB

Collection: Analytics_Hive

Documents: Filter by id

ID
e832bc60-170c-488b-bf1e-bd3adb826536
16e834ad-a228-48f4-b52f-8d8075732287
79ba821c-5aa9-48d0-ab0d-68dc4df4604d
ca8e6078-80b1-44b3-8d06-52f8d8c28ea6

e832bc60-170c-488b-bf1e-bd3adb826536
DOCUMENT

```
{
  "id": "e832bc60-170c-488b-bf1e-bd3adb826536",
  "minute": 23,
  "count": 679,
  "month": 2,
  "day": 16,
  "hour": 19
}
```

Step 4: Run a Pig job using Cosmos DB 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 Cosmos DB 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 Cosmos DB. 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 Cosmos DB.
$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
```

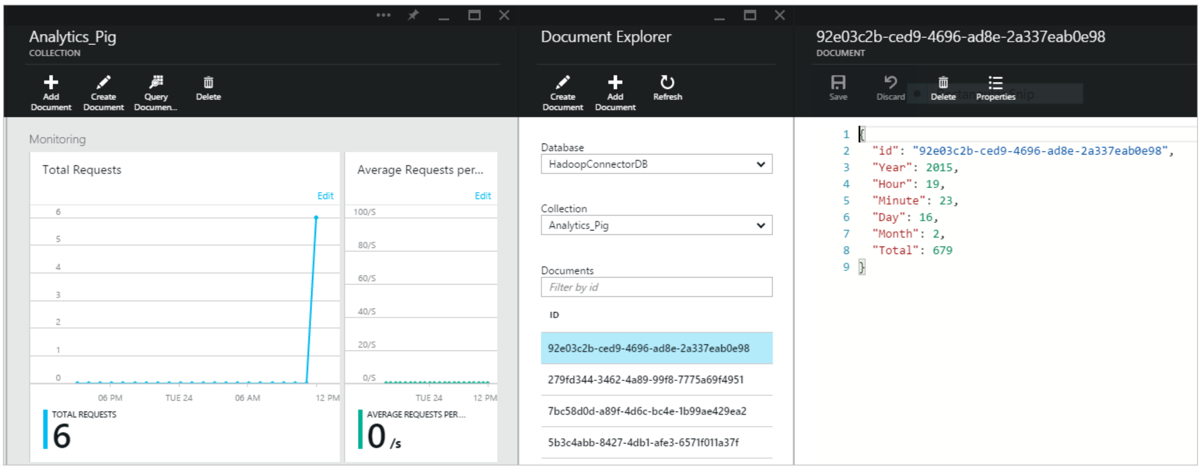
8. Add the following to print the standard output and the start and end times.

```
# 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.



Database: HadoopConnectorDB

Collection: Analytics_Pig

Documents: Filter by id

ID
92e03c2b-ced9-4696-ad8e-2a337eab0e98
279fd344-3462-4a89-99f8-7775a69f4951
7bc58d0d-a89f-4d6c-bc4e-1b99ae429ea2
5b3c4abb-8427-4db1-afe3-6571f011a37f

```
{
  "id": "92e03c2b-ced9-4696-ad8e-2a337eab0e98",
  "Year": 2015,
  "Hour": 19,
  "Minute": 23,
  "Day": 16,
  "Month": 2,
  "Total": 679
}
```

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 Cosmos DB 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 asynchronized call. To check the completion of the job, use the *Wait-AzureHDInsightJob* cmdlet.

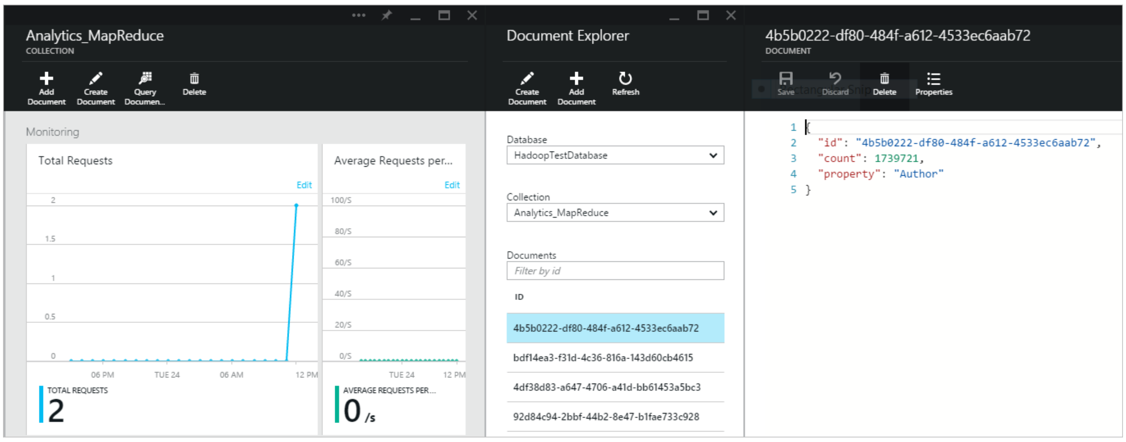
4. Add the following command to check any errors with running the MapReduce job.

```
# 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.

a. Click **Browse** on the left-side panel.
b. Click **everything** at the top-right of the browse panel.
c. Find and click **Cosmos DB Accounts**.
d. Next, find your **Cosmos DB Account**, then **Cosmos DB Database** and your **DocumentDB Collection** associated with the output collection specified in your MapReduce job.
e. Finally, click **Document Explorer** underneath **Developer Tools**.

   You will see the results of your MapReduce job.



Database: HadoopTestDatabase

Collection: Analytics_MapReduce

Documents: Filter by id

ID
4b5b0222-df80-484f-a612-4533ec6aab72
bdf14ea3-f31d-4c36-816a-143d60cb4615
4df38d83-a647-4706-a41d-bb61453a5bc3
92d84c94-2bbf-44b2-8e47-b1fae733c928

```
{
  "id": "4b5b0222-df80-484f-a612-4533ec6aab72",
  "count": 1739721,
  "property": "Author"
}
```

Next Steps

Congratulations! You just ran your first Hive, Pig, and MapReduce jobs using Azure Cosmos DB 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 Cosmos DB with Azure Search using indexers

5/10/2017 • 6 min to read • Edit Online

If you want to implement a great search experience over your Cosmos DB data, you can use an Azure Search indexer to pull data into an Azure Search index. In this article, we show you how to integrate Azure Cosmos DB with Azure Search without having to write any code to maintain indexing infrastructure.

To set up a Cosmos DB indexer, you must have an Azure Search service, and create an index, datasource, and finally the indexer. You can create these objects using the portal, .NET SDK, or REST API for all non-.NET languages.

If you opt for the portal, the Import data wizard guides you through the creation of all these resources.

> NOTE
>
> Cosmos DB is the next generation of DocumentDB. Although the product name is changed, syntax is the same as before. Please continue to specify `documentdb` as directed in this indexer article.

> TIP
>
> You can launch the **Import data** wizard from the Cosmos DB dashboard to simplify indexing for that data source. In left-navigation, go to **Collections** > **Add Azure Search** to get started.

Azure Search indexer concepts

Azure Search supports the creation and management of data sources (including Cosmos DB) and indexers that operate against those data sources.

A **data source** specifies the data to index, credentials, and policies for identifying 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. An indexer can be 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

To create a data source, do a POST:

```
POST https://[service name].search.windows.net/datasources?api-version=2016-09-01
Content-Type: application/json
api-key: [Search service admin key]

{
   "name": "mydocdbdatasource",
   "type": "documentdb",
   "credentials": {
      "connectionString":
"AccountEndpoint=https://myDocDbEndpoint.documents.azure.com;AccountKey=myDocDbAuthKey;Database=myDocDbDatabaseId"
   },
   "container": { "name": "myDocDbCollectionId", "query": null },
   "dataChangeDetectionPolicy": {
      "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
      "highWaterMarkColumnName": "_ts"
   }
}
```

The body of the request contains the data source definition, which should include the following fields:

- **name**: Choose any name to represent your Cosmos DB database.
- **type**: Must be `documentdb`.
- **credentials**:
    - **connectionString**: Required. Specify the connection info to your Azure Cosmos DB database in the following format: `AccountEndpoint=<Cosmos DB endpoint url>;AccountKey=<Cosmos DB auth key>;Database=<Cosmos DB database id>`
- **container**:
    - **name**: Required. Specify the id of the Cosmos DB 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**: Recommended. See Indexing Changed Documents section.
- **dataDeletionDetectionPolicy**: Optional. See Indexing Deleted Documents section.

Using queries to shape indexed data

You can specify a Cosmos DB query to flatten nested properties or arrays, project JSON properties, and filter the data to be indexed.

Example document:

```
{
   "userId": 10001,
   "contact": {
      "firstName": "andy",
      "lastName": "hoh"
   },
   "company": "microsoft",
   "tags": ["azure", "documentdb", "search"]
}
```

Filter query:

```
SELECT * FROM c WHERE c.company = "microsoft" and c._ts >= @HighWaterMark
```

Flattening query:

```
SELECT c.id, c.userId, c.contact.firstName, c.contact.lastName, c.company, c._ts FROM c WHERE c._ts >= @HighWaterMark
```

Projection query:

```
SELECT VALUE { "id":c.id, "Name":c.contact.firstName, "Company":c.company, "_ts":c._ts } FROM c WHERE c._ts >= @HighWaterMark
```

Array flattening query:

```
SELECT c.id, c.userId, tag, c._ts FROM c JOIN tag IN c.tags WHERE c._ts >= @HighWaterMark
```

Step 2: Create an index

Create a target Azure Search index if you don’t have one already. You can create an index using the Azure portal UI, the Create Index REST API or Index class.

The following example creates an index with an id and description field:

```
POST https://[service name].search.windows.net/indexes?api-version=2016-09-01
Content-Type: application/json
api-key: [Search service admin key]

{
  "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
  }]
}
```

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 Cosmos DB's `_rid` property, which gets renamed to `rid` in Azure Search. Also, Cosmos DB's `_rid` values contain characters that are invalid in Azure Search keys. For this reason, the `_rid` values are Base64 encoded.

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

JSON DATA TYPE	COMPATIBLE TARGET INDEX FIELD TYPES
Numbers that look like floating-points	Edm.Double, Edm.String
String	Edm.String
Arrays of primitive types, for example ["a", "b", "c"]	Collection(Edm.String)
Strings that look like dates	Edm.DateTimeOffset, Edm.String
GeoJSON objects, for example { "type": "Point", "coordinates": [long, lat] }	Edm.GeographyPoint
Other JSON objects	N/A

Step 3: Create an indexer

Once the index and data source have been created, you're ready to create the indexer:

```
POST https://[service name].search.windows.net/indexers?api-version=2016-09-01
Content-Type: application/json
api-key: [admin key]

{
  "name" : "mydocdbindexer",
  "dataSourceName" : "mydocdbdatasource",
  "targetIndexName" : "mysearchindex",
  "schedule" : { "interval" : "PT2H" }
}
```

This indexer runs every two hours (schedule interval is set to "PT2H"). To run an indexer every 30 minutes, set the interval to "PT30M". The shortest supported interval is 5 minutes. The schedule is optional - if omitted, an indexer runs only once when it's created. However, you can run an indexer on-demand at any time.

For more details on the Create Indexer API, check out Create Indexer.

Running indexer on-demand

In addition to running periodically on a schedule, an indexer can also be invoked on demand:

```
POST https://[service name].search.windows.net/indexers/[indexer name]/run?api-version=2016-09-01
api-key: [Search service admin key]
```

> NOTE
>
> When Run API returns successfully, the indexer invocation has been scheduled, but the actual processing happens asynchronously.

You can monitor the indexer status in the portal or using the Get Indexer Status API, which we describe next.

Getting indexer status

You can retrieve the status and execution history of an indexer:

```
GET https://[service name].search.windows.net/indexers/[indexer name]/status?api-version=2016-09-01
api-key: [Search service admin key]
```

The response contains overall indexer status, the last (or in-progress) indexer invocation, and the history of recent indexer invocations.

```
{
  "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).

Indexing 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` (timestamp) property provided by Cosmos DB, which is specified as follows:

```
{
  "@odata.type" : "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
  "highWaterMarkColumnName" : "_ts"
}
```

Using this policy is highly recommended to ensure good indexer performance.

If you are using a custom query, make sure that the `_ts` property is projected by the query.

Indexing deleted documents

When rows are deleted from the collection, you normally want to 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"
}
```

If you are using a custom query, make sure that the property referenced by `softDeleteColumnName` is projected by the query.

The following example creates a data source with a soft-deletion policy:

```
POST https://[Search service name].search.windows.net/datasources?api-version=2016-09-01
Content-Type: application/json
api-key: [Search service admin key]

{
  "name": "mydocdbdatasource",
  "type": "documentdb",
  "credentials": {
    "connectionString":
"AccountEndpoint=https://myDocDbEndpoint.documents.azure.com;AccountKey=myDocDbAuthKey;Database=myDocDbDatabaseId"
  },
  "container": { "name": "myDocDbCollectionId" },
  "dataChangeDetectionPolicy": {
    "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
    "highWaterMarkColumnName": "_ts"
  },
  "dataDeletionDetectionPolicy": {
    "@odata.type": "#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
    "softDeleteColumnName": "isDeleted",
    "softDeleteMarkerValue": "true"
  }
}
```

Next steps

Congratulations! You have learned how to integrate Azure Cosmos DB with Azure Search using the indexer for Cosmos DB.

- To learn how more about Azure Cosmos DB, see the Cosmos DB service page.
- To learn how more about Azure Search, see the Search service page.

Move data to and from Azure Cosmos DB using Azure Data Factory

5/11/2017 • 11 min to read • Edit Online

This article explains how to use the Copy Activity in Azure Data Factory to move data to/from Azure Cosmos DB (DocumentDB API). It builds on the Data Movement Activities article, which presents a general overview of data movement with the copy activity.

You can copy data from any supported source data store to Azure Cosmos DB or from Azure Cosmos DB to any supported sink data store. For a list of data stores supported as sources or sinks by the copy activity, see the Supported data stores table.

> IMPORTANT
>
> Azure Cosmos DB connector only support DocumentDB API.

To copy data as-is to/from JSON files or another Cosmos DB collection, see Import/Export JSON documents.

Getting started

You can create a pipeline with a copy activity that moves data to/from Azure Cosmos DB by using different tools/APIs.

The easiest way to create a pipeline is to use the **Copy Wizard**. See Tutorial: Create a pipeline using Copy Wizard for a quick walkthrough on creating a pipeline using the Copy data wizard.

You can also use the following tools to create a pipeline: **Azure portal**, **Visual Studio**, **Azure PowerShell**, **Azure Resource Manager template**, **.NET API**, and **REST API**. See Copy activity tutorial for step-by-step instructions to create a pipeline with a copy activity.

Whether you use the tools or APIs, you perform the following steps to create a pipeline that moves data from a source data store to a sink data store:

1. Create **linked services** to link input and output data stores to your data factory.
2. Create **datasets** to represent input and output data for the copy operation.
3. Create a **pipeline** with a copy activity that takes a dataset as an input and a dataset as an output.

When you use the wizard, JSON definitions for these Data Factory entities (linked services, datasets, and the pipeline) are automatically created for you. When you use tools/APIs (except .NET API), you define these Data Factory entities by using the JSON format. For samples with JSON definitions for Data Factory entities that are used to copy data to/from Cosmos DB, see JSON examples section of this article.

The following sections provide details about JSON properties that are used to define Data Factory entities specific to Cosmos DB:

Linked service properties

The following table provides description for JSON elements specific to Azure Cosmos DB linked service.

PROPERTY	DESCRIPTION	REQUIRED
type	The type property must be set to: **DocumentDb**	Yes
connectionString	Specify information needed to connect to Azure Cosmos DB database.	Yes

Dataset 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 Cosmos DB document collection.	Yes

Example:

```
{
 "name": "PersonCosmosDbTable",
 "properties": {
  "type": "DocumentDbCollection",
  "linkedServiceName": "CosmosDbLinkedService",
  "typeProperties": {
   "collectionName": "Person"
  },
  "external": true,
  "availability": {
   "frequency": "Day",
   "interval": 1
  }
 }
}
```

Schema by Data Factory

For schema-free data stores such as Azure Cosmos DB, 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.

Copy activity 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 Azure Cosmos DB. Example: `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\"`	No If not specified, the SQL statement that is executed: `select <columns defined in structure> from mycollection`
nestingSeparator	Special character to indicate that the document is nested	Any character. Azure Cosmos DB 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	A special character in the source column name to indicate that nested document is needed. For example above: `Name.First` in the output table produces the following JSON structure in the Cosmos DB document: "Name": { "First": "John" },	Character that is used to separate nesting levels. Default value is `.` (dot).	Character that is used to separate nesting levels. Default value is `.` (dot).
writeBatchSize	Number of parallel requests to Azure Cosmos DB service to create documents. You can fine-tune the performance when copying data to/from Cosmos DB by using this property. You can expect a better performance when you increase writeBatchSize because more parallel requests to Cosmos DB are sent. However you'll need to avoid throttling that can throw the error message: "Request rate is large". 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).	Integer	No (default: 5)
writeBatchTimeout	Wait time for the operation to complete before it times out.	timespan Example: "00:30:00" (30 minutes).	No

Import/Export JSON documents

Using this Cosmos DB connector, you can easily

- Import JSON documents from various sources into Cosmos DB, including Azure Blob, Azure Data Lake, on-premises File System or other file-based stores supported by Azure Data Factory.
- Export JSON documents from Cosmos DB collecton into various file-based stores.
- Migrate data between two Cosmos DB collections as-is.

To achieve such schema-agnostic copy,

- When using copy wizard, check the **"Export as-is to JSON files or Cosmos DB collection"** option.
- When using JSON editing, do not specify the "structure" section in Cosmos DB dataset(s) nor "nestingSeparator" property on Cosmos DB source/sink in copy activity. To import from/export to JSON files, in the file store dataset specify format type as "JsonFormat", config "filePattern" and skip the rest format settings, see JSON format section on details.

JSON examples

The following examples provide sample JSON definitions that you can use to create a pipeline by using Azure portal or Visual Studio or Azure PowerShell. They show how to copy data to and from Azure Cosmos DB and Azure Blob Storage. However, data can be copied **directly** from any of the sources to any of the sinks stated here using the Copy Activity in Azure Data Factory.

Example: Copy data from Azure Cosmos DB 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 Cosmos DB to Azure Blob. The JSON properties used in these samples are described in sections following the samples.

Azure Cosmos DB linked service:

```
{
  "name": "CosmosDbLinkedService",
  "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 Cosmos DB 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": "PersonCosmosDbTable",
  "properties": {
    "type": "DocumentDbCollection",
    "linkedServiceName": "CosmosDbLinkedService",
    "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 Cosmos DB database:

```
{
  "PersonId": 2,
  "Name": {
    "First": "Jane",
    "Middle": "",
    "Last": "Doe"
  }
}
```

Cosmos DB 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 Azure Cosmos DB 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": "PersonCosmosDbTable"
     }
    ],
    "outputs": [
     {
      "name": "PersonBlobTableOut"
     }
    ],
    "policy": {
     "concurrency": 1
    },
    "name": "CopyFromDocDbToBlob"
   }
  ],
  "start": "2015-04-01T00:00:00Z",
  "end": "2015-04-02T00:00:00Z"
 }
}
```

Example: Copy data from Azure Blob to Azure Cosmos DB

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 Cosmos DB. 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 Cosmos DB linked service:

```
{
  "name": "CosmosDbLinkedService",
  "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 Cosmos DB output dataset:

The sample copies data to a collection named “Person”.

```
{
  "name": "PersonCosmosDbTableOut",
  "properties": {
    "structure": [
      {
        "name": "Id",
        "type": "Int"
      },
      {
        "name": "Name.First",
        "type": "String"
      },
      {
        "name": "Name.Middle",
        "type": "String"
      },
      {
        "name": "Name.Last",
        "type": "String"
      }
    ],
    "type": "DocumentDbCollection",
    "linkedServiceName": "CosmosDbLinkedService",
    "typeProperties": {
      "collectionName": "Person"
    },
    "availability": {
      "frequency": "Day",
      "interval": 1
    }
  }
}
```

The following pipeline copies data from Azure Blob to the Person collection in the Cosmos DB. 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": "PersonCosmosDbTableOut"
          }
        ],
        "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 Cosmos DB will be as:

```
{
  "Id": 1,
  "Name": {
    "First": "John",
    "Middle": null,
    "Last": "Doe"
  },
  "id": "a5e8595c-62ec-4554-a118-3940f4ff70b6"
}
```

Azure Cosmos DB 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.

Appendix

1. **Question:** Does the Copy Activity support update of existing records?

   Answer: No.

2. **Question:** How does a retry of a copy to Azure Cosmos DB 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 Azure Cosmos DB 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

5/10/2017 • 16 min to read • Edit Online

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?

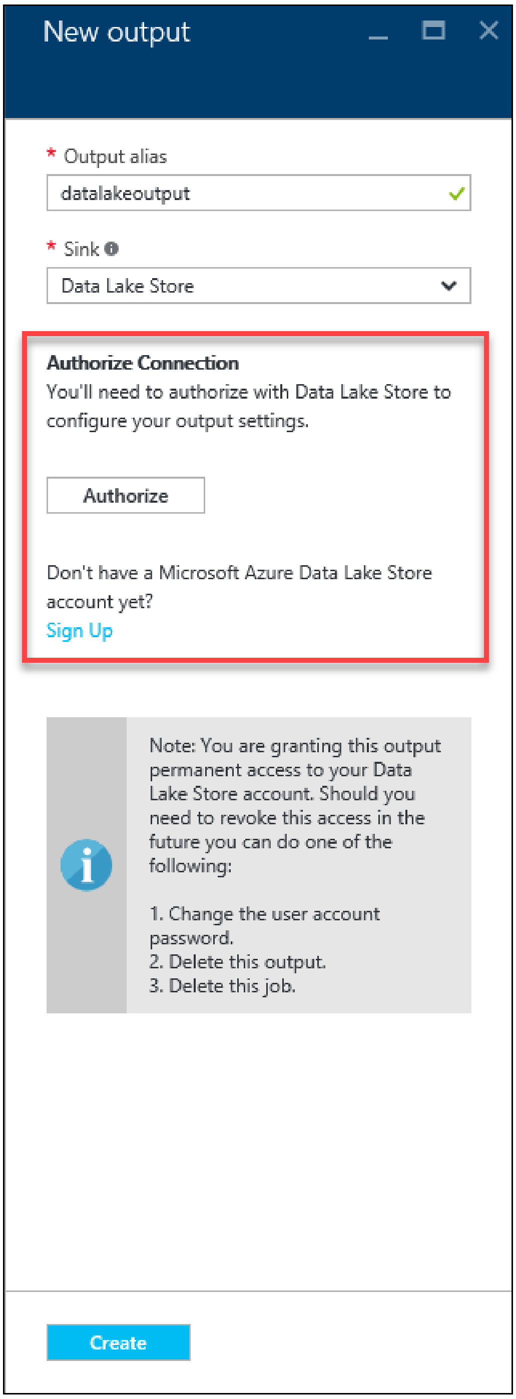
In order to enable a variety of application patterns, Azure Stream Analytics has different options for storing output and viewing analysis results. This makes it easy to view job output and gives you flexibility in the consumption and storage of the job output for data warehousing and other purposes. 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. 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 (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?

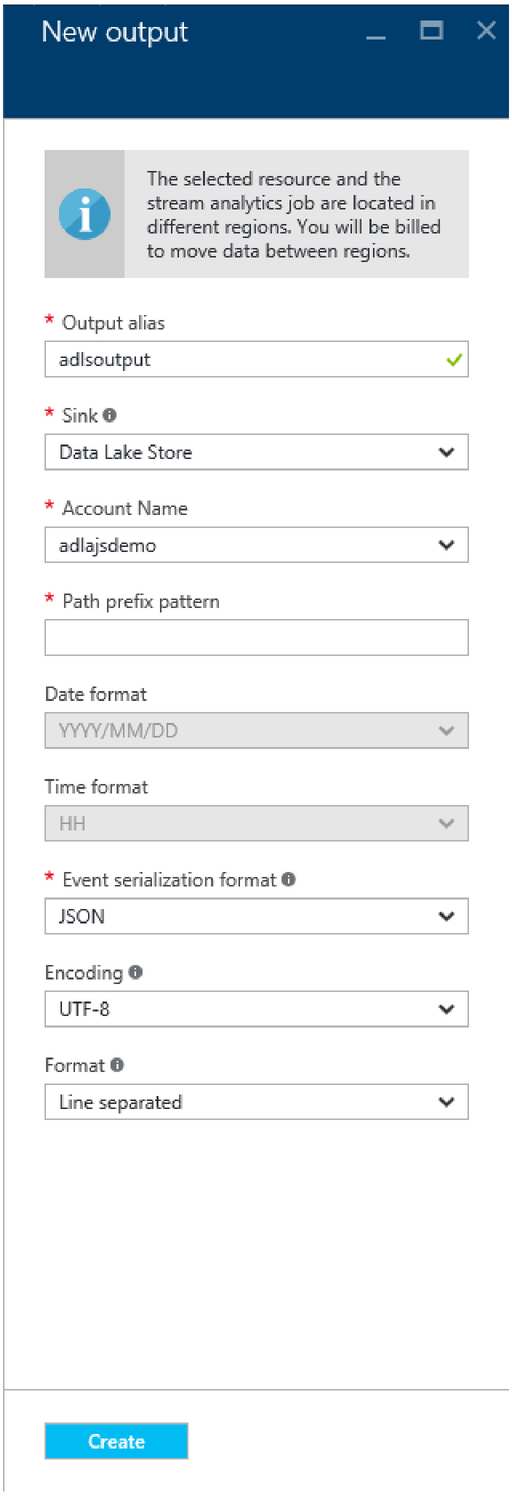
Sign Up

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

The table below lists the property names and their description needed for creating a Data Lake Store output.

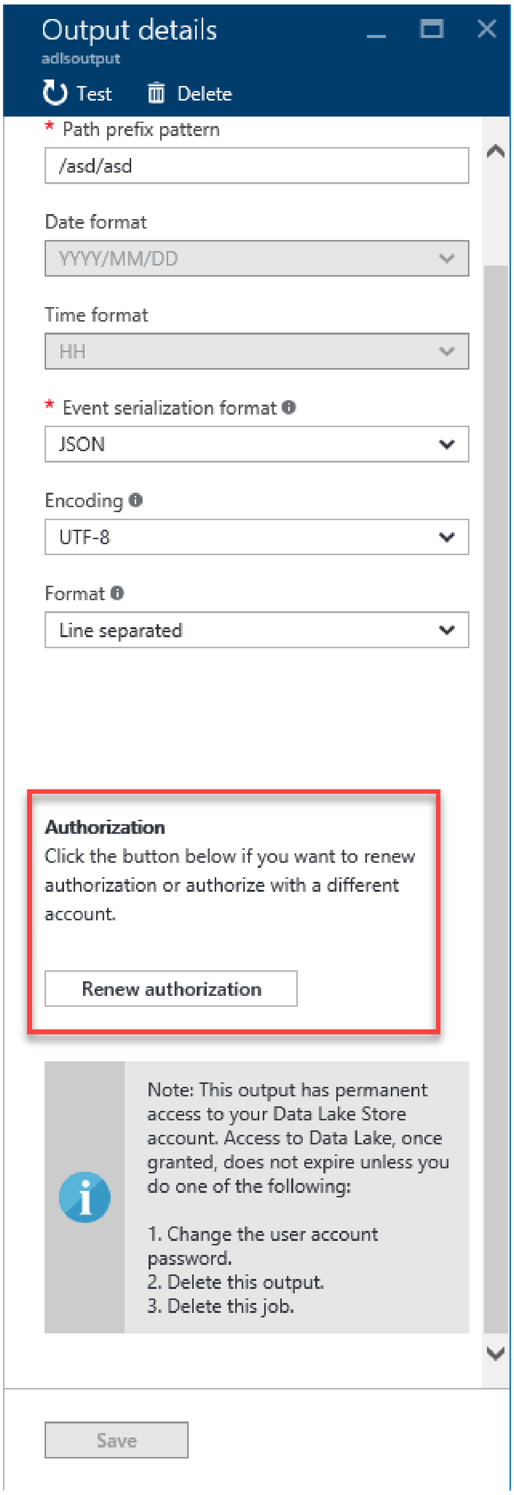
PROPERTY NAME	DESCRIPTION
Output Alias	This is a friendly name used in queries to direct the query output to this Data Lake Store.
Account Name	The name of the Data Lake Storage account where you are sending your output. You will be presented with a drop down list of Data Lake Store accounts to which the user logged in to the portal has access to.

Path Prefix Pattern [*optional*]	The file path used to write your files within the specified Data Lake Store Account.
	{date}, {time}
	Example 1: folder1/logs/{date}/{time}
	Example 2: folder1/logs/{date}

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.

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

* 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.

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.

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]	The file path used to write your blobs within the specified container.
	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:
	{date}, {time}
	Example 1: cluster1/logs/{date}/{time}
	Example 2: cluster1/logs/{date}

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.

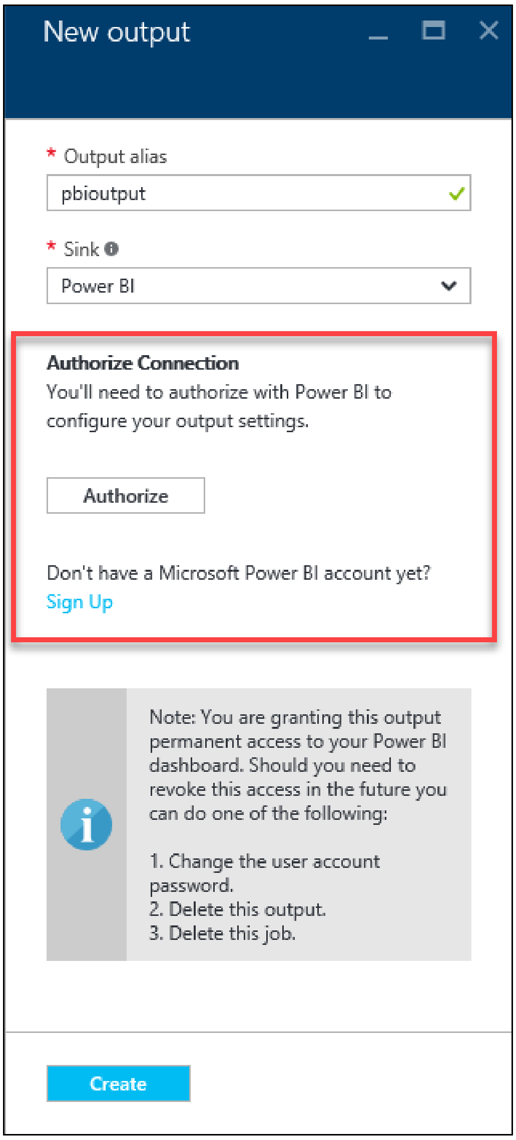
PROPERTY NAME	DESCRIPTION
Format	Only applicable for JSON type. Line separated specifies that the output will be formatted by having each JSON object separated by a new line. Array specifies that the output will be formatted as an array of JSON objects.

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

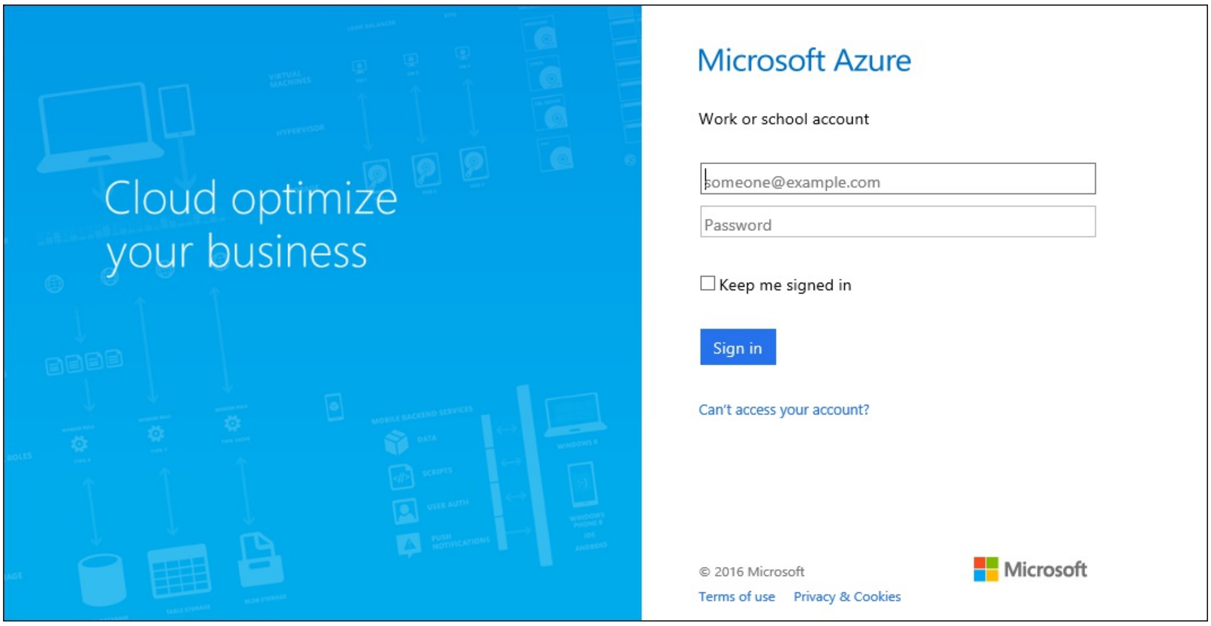
Don't have a Microsoft Power BI account yet?
Sign Up

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.

Schema Creation

Azure Stream Analytics creates a Power BI dataset and table on behalf of the user if one does not already exist. In all other cases, the table is updated with new values.Currently, there is a the limitation that only one table can exist within a dataset.

Data type conversion from ASA to Power BI

Azure Stream Analytics updates the data model dynamically at runtime if the output schema changes. Column name changes, column type changes, and the addition or removal of columns are all tracked.

This table covers the data type conversions from Stream Analytics data types to Power BIs Entity Data Model (EDM) types if a POWER BI dataset and table do not exist.

FROM STREAM ANALYTICS	TO POWER BI
bigint	Int64
nvarchar(max)	String
datetime	Datetime
float	Double
Record array	String type, Constant value “IRecord” or “IArray”

Schema Update

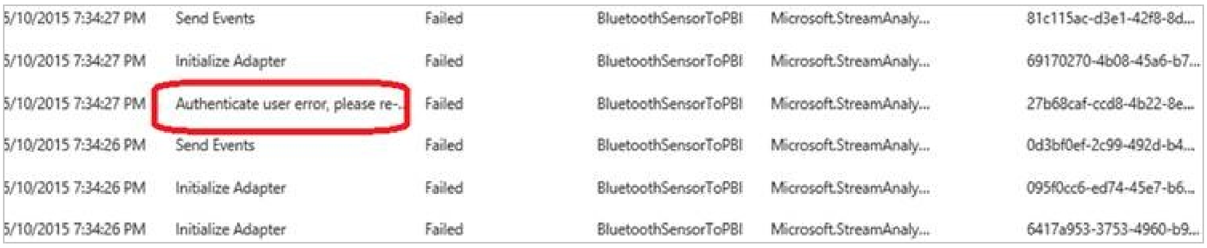
Stream Analytics infers the data model schema based on the first set of events in the output. Later, if necessary, the data model schema is updated to accommodate incoming events that may not fit into the original schema.

The `SELECT *` query should be avoided to prevent dynamic schema update across rows. In addition to potential performance implications, it could also result in indeterminacy of the time taken for the results. The exact fields that need to be shown on Power BI dashboard should be selected. Additionally, the data values should be compliant with the chosen data type.

PREVIOUS/CURRENT	INT64	STRING	DATETIME	DOUBLE
Int64	Int64	String	String	Double
Double	Double	String	String	Double
String	String	String	String	
Datetime	String	String	Datetime	String

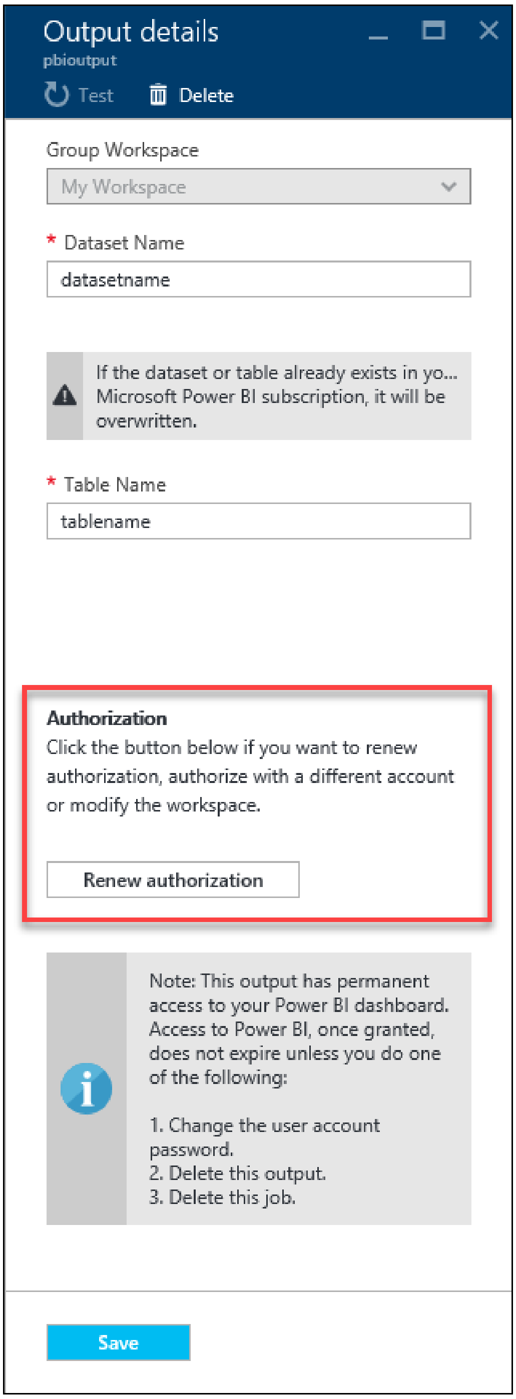
Renew Power BI Authorization

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" 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

pbioutput

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

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.

PROPERTY NAME	DESCRIPTION
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.
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

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.
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
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.

Azure Cosmos DB

Azure Cosmos DB 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 below list details the property names and their description for creating an Azure Cosmos DB output.

- **Output Alias** – An alias to refer this output in your ASA query
- **Account Name** – The name or endpoint URI of the Cosmos DB account.
- **Account Key** – The shared access key for the Cosmos DB account.
- **Database** – The Cosmos DB database name.
- **Collection Name Pattern** – The collection name or their pattern for the collections to be used. The collection name format can be constructed using the optional {partition} token, where partitions start from 0. Following are sample valid inputs:
  1) MyCollection – One collection named "MyCollection" must exist.
  2) MyCollection{partition} – Such collections must exist– "MyCollection0", "MyCollection1", "MyCollection2" and so on.
- **Partition Key** – Optional. This is only needed if you are using a {parition} token in your collection name pattern. The name of the field in output events used to specify the key for partitioning output across collections. For single collection output, any arbitrary output column can be used e.g. PartitionId.
- **Document ID** – Optional. The name of the field in output events used to specify the primary key on which insert or update operations are based.

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

Notifying patients of HL7 FHIR health care record changes using Logic Apps and Azure Cosmos DB

6/6/2017 • 4 min to read • Edit Online

Azure MVP Howard Edidin was recently contacted by a healthcare organization that wanted to add new functionality to their patient portal. They needed to send notifications to patients when their health record was updated, and they needed patients to be able to subscribe to these updates.

This article walks through the change feed notification solution created for this healthcare organization using Azure Cosmos DB, Logic Apps, and Service Bus.

Project requirements

- Providers send HL7 Consolidated-Clinical Document Architecture (C-CDA) documents in XML format. C-CDA documents encompass just about every type of clinical document, including clinical documents such as family histories and immunization records, as well as administrative, workflow, and financial documents.
- C-CDA documents are converted to HL7 FHIR Resources in JSON format.
- Modified FHIR resource documents are sent by email in JSON format.

Solution workflow

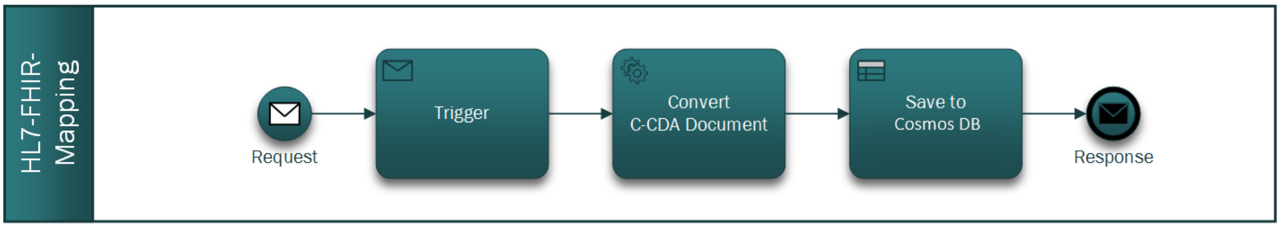
At a high level, the project required the following workflow steps:

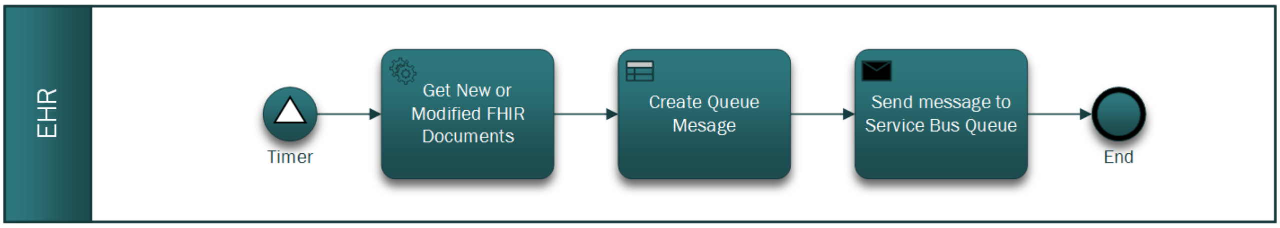
1. Convert C-CDA documents to FHIR resources.
2. Perform recurring trigger polling for modified FHIR resources.
3. Call a custom app, FhirNotificationApi, to connect to Azure Cosmos DB and query for new or modified documents.
4. Save the response to to the Service Bus queue.
5. Poll for new messages in the Service Bus queue.
6. Send email notifications to patients.

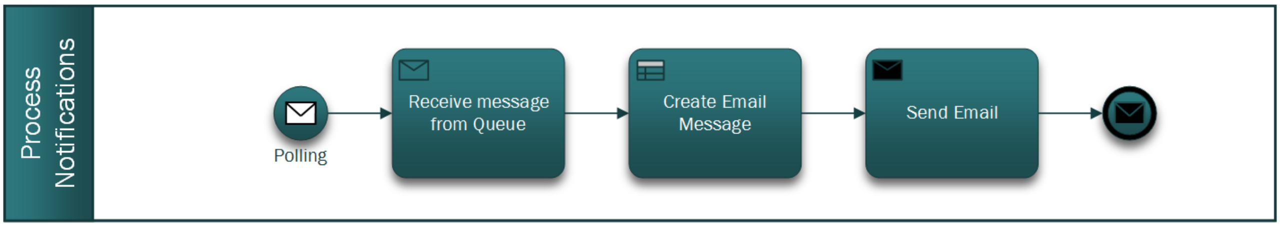
Solution architecture

This solution requires three Logic Apps to meet the above requirements and complete the solution workflow. The three logic apps are:

1. **HL7-FHIR-Mapping app**: Receives the HL7 C-CDA document, transforms it to the FHIR Resource, then saves it to Azure Cosmos DB.
2. **EHR app**: Queries the Azure Cosmos DB FHIR repository and saves the response to a Service Bus queue. This logic app uses an API app to retrieve new and changed documents.
3. **Process notification app**: Sends an email notification with the FHIR resource documents in the body.



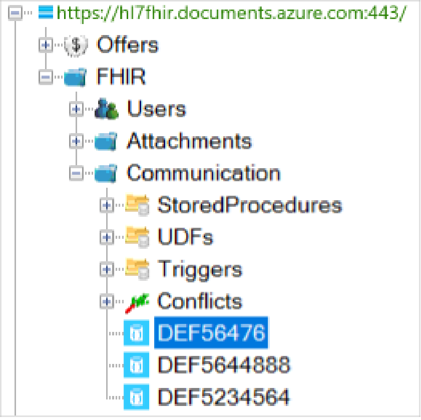

EHR
Timer
Get New or Modified FHIR Documents
Create Queue Mesage
Send message to Service Bus Queue
End



Azure services used in the solution

Azure Cosmos DB DocumentDB API

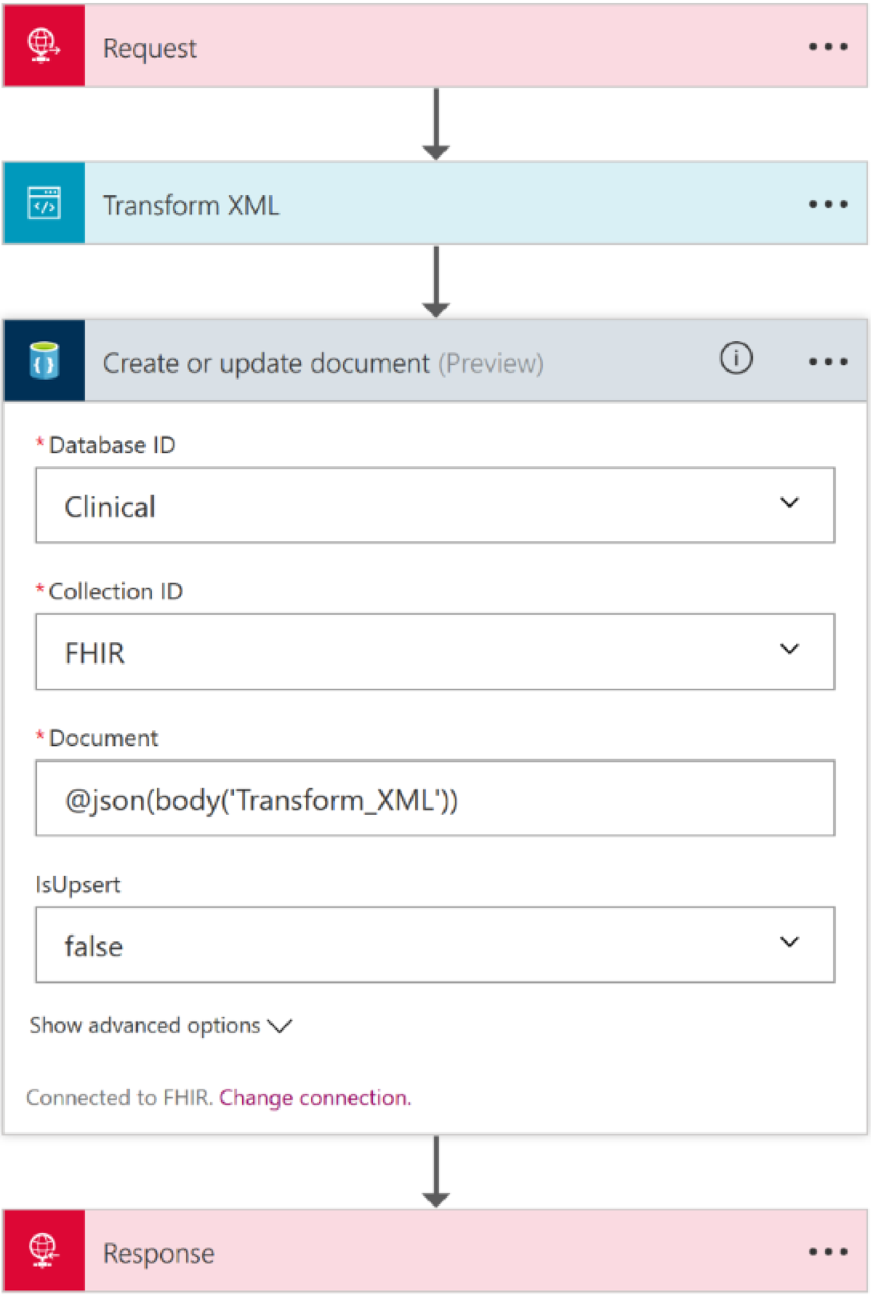
Azure Cosmos DB is the repository for the FHIR resources as shown in the following figure.



Logic Apps

Logic Apps handle the workflow process. The following screenshots show the Logic apps created for this solution.

1. **HL7-FHIR-Mapping app**: Receive the HL7 C-CDA document and transform it to an FHIR resource using the Enterprise Integration Pack for Logic Apps. The Enterprise Integration Pack handles the mapping from the C-CDA to FHIR resources.



Request

Transform XML

Create or update document (Preview)

* Database ID

Clinical

* Collection ID

FHIR

* Document

@json(body('Transform_XML'))

IsUpsert

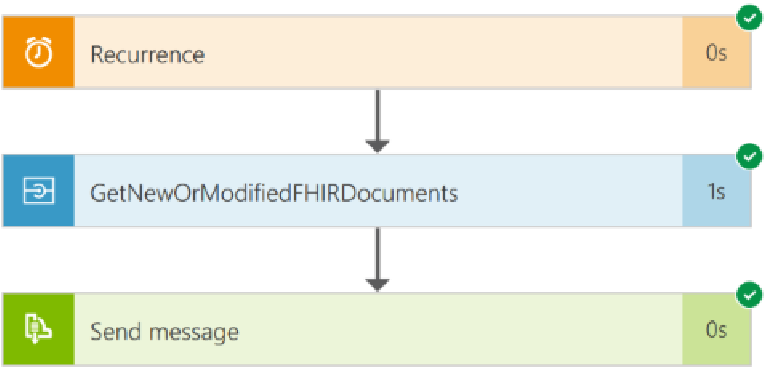
false

Show advanced options

Connected to FHIR. Change connection.

Response

2. **EHR app**: Query the Azure Cosmos DB FHIR repository and save the response to a Service Bus queue. The code for the GetNewOrModifiedFHIRDocuments app is below.

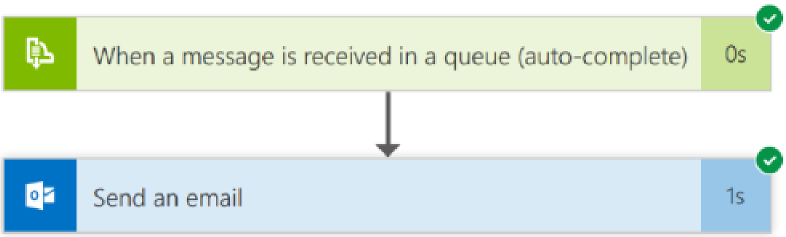


Recurrence 0s

GetNewOrModifiedFHIRDocuments 1s

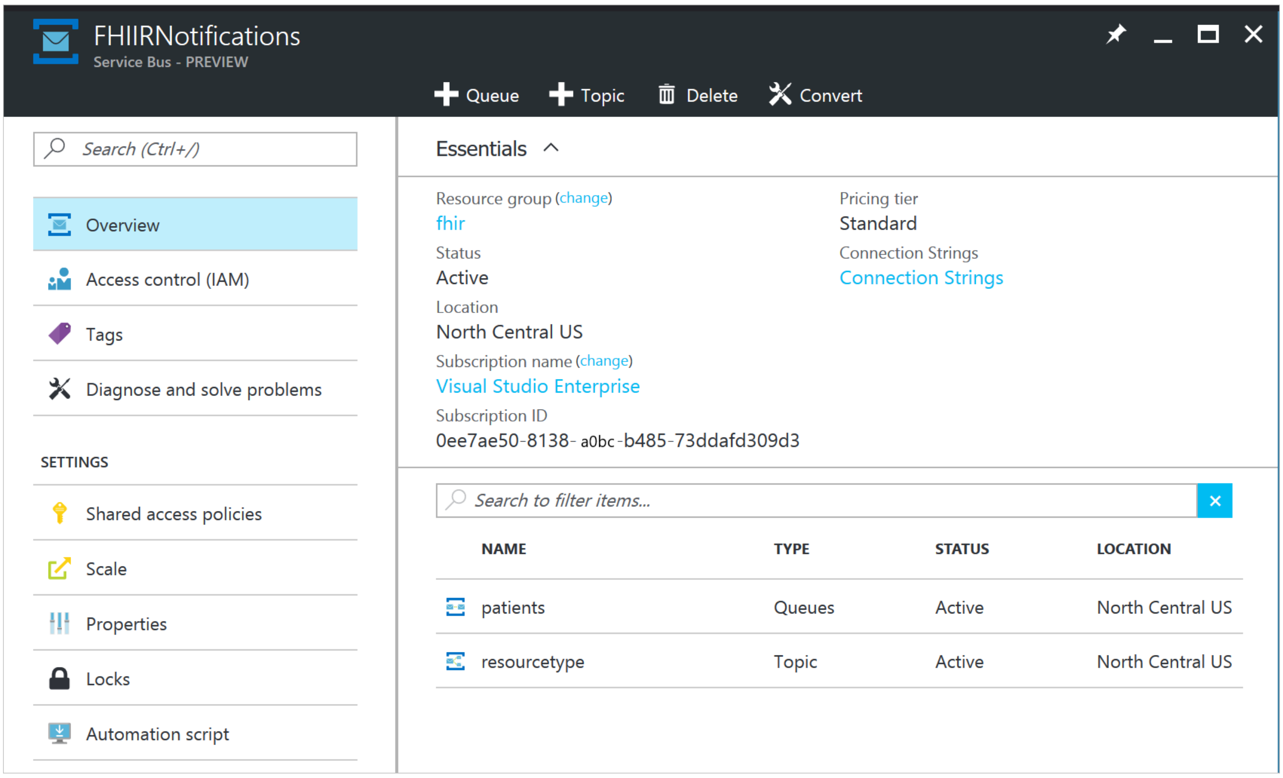
Send message 0s

3. **Process notification app**: Send an email notification with the FHIR resource documents in the body.



Service Bus

The following figure shows the patients queue. The Tag property value is used for the email subject.



Resource group (change)
fhir

Status
Active

Location
North Central US

Subscription name (change)
Visual Studio Enterprise

Subscription ID
0ee7ae50-8138- a0bc -b485-73ddafd309d3

Pricing tier
Standard

Connection Strings
Connection Strings

NAME	TYPE	STATUS	LOCATION
patients	Queues	Active	North Central US
resourcetype	Topic	Active	North Central US

API app

An API app connects to Azure Cosmos DB and queries for new or modified FHIR documents By resource type. This app has one controller, **FhirNotificationApi** with a one operation **GetNewOrModifiedFhirDocuments**, see source for API app.

We are using the `CreateDocumentChangeFeedQuery` class from the Azure Cosmos DB DocumentDB .NET API. For more information, see the change feed article.

GetNewOrModifiedFhirDocuments operation

Inputs

- DatabaseId
- CollectionId
- HL7 FHIR Resource Type name
- Boolean: Start from Beginning
- Int: Number of documents returned

Outputs

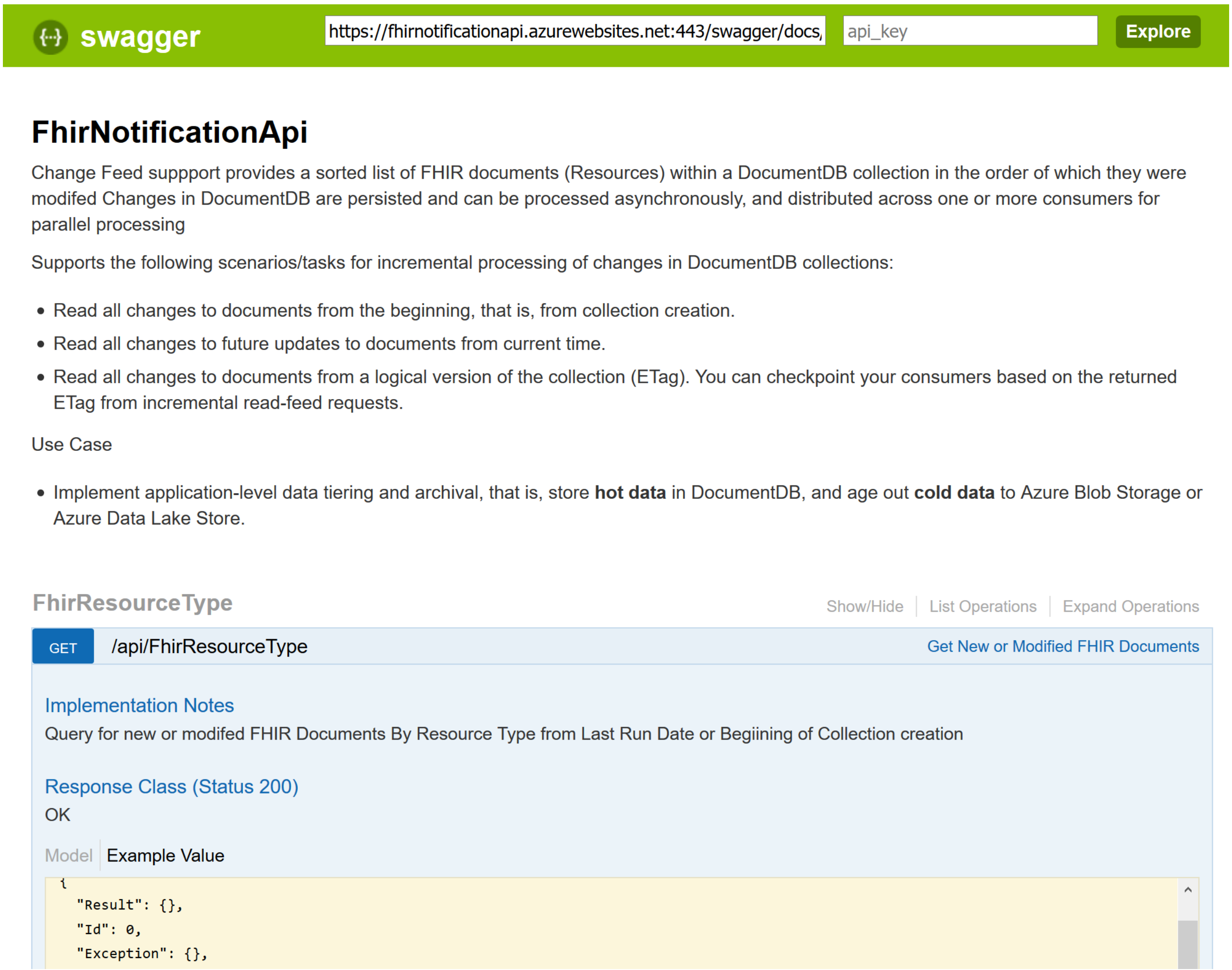
- Success: Status Code: 200, Response: List of Documents (JSON Array)
- Failure: Status Code: 404, Response: "No Documents found for '*resource name*' Resource Type"

Source for the API app

```
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using System.Web.Http;
using Microsoft.Azure.Documents;
using Microsoft.Azure.Documents.Client;
using Swashbuckle.Swagger.Annotations;
using TRex.Metadata;

namespace FhirNotificationApi.Controllers
{
    /// <summary>
    ///     FHIR Resource Type Controller
    /// </summary>
    /// <seealso cref="System.Web.Http.ApiController" />
    public class FhirResourceTypeController : ApiController
    {
        /// <summary>
        ///     Gets the new or modified FHIR documents from Last Run Date
        ///     or create date of the collection
        /// </summary>
        /// <param name="databaseId"></param>
        /// <param name="collectionId"></param>
        /// <param name="resourceType"></param>
        /// <param name="startfromBeginning"></param>
        /// <param name="maximumItemCount">-1 returns all (default)</param>
        /// <returns></returns>
        [Metadata("Get New or Modified FHIR Documents",
            "Query for new or modifed FHIR Documents By Resource Type " +
            "from Last Run Date or Begining of Collection creation"
        )]
        [SwaggerResponse(HttpStatusCode.OK, type: typeof(Task<dynamic>))]
        [SwaggerResponse(HttpStatusCode.NotFound, "No New or Modifed Documents found")]
        [SwaggerOperation("GetNewOrModifiedFHIRDocuments")]
        public async Task<dynamic> GetNewOrModifiedFhirDocuments(
            [Metadata("Database Id", "Database Id")] string databaseId,
            [Metadata("Collection Id", "Collection Id")] string collectionId,
            [Metadata("Resource Type", "FHIR resource type name")] string resourceType,
            [Metadata("Start from Beginning ", "Change Feed Option")] bool startfromBeginning,
            [Metadata("Maximum Item Count", "Number of documents returned. '-1 returns all' (default)")] int maximumItemCount = -1
        )
        {
            var collectionLink = UriFactory.CreateDocumentCollectionUri(databaseId, collectionId);

            var context = new DocumentDbContext();

            var docs = new List<dynamic>();

            var partitionKeyRanges = new List<PartitionKeyRange>();
            FeedResponse<PartitionKeyRange> pkRangesResponse;

            do
            {
                pkRangesResponse = await context.Client.ReadPartitionKeyRangeFeedAsync(collectionLink);
                partitionKeyRanges.AddRange(pkRangesResponse);
            } while (pkRangesResponse.ResponseContinuation != null);

            foreach (var pkRange in partitionKeyRanges)
            {
                var changeFeedOptions = new ChangeFeedOptions
                {
                    StartFromBeginning = startfromBeginning,
                    RequestContinuation = null,
                    MaxItemCount = maximumItemCount,
                    PartitionKeyRangeId = pkRange.Id
                };
        };

        using (var query = context.Client.CreateDocumentChangeFeedQuery(collectionLink, changeFeedOptions))
        {
          do
          {
            if (query != null)
            {
              var results = await query.ExecuteNextAsync<dynamic>().ConfigureAwait(false);
              if (results.Count > 0)
                docs.AddRange(results.Where(doc => doc.resourceType == resourceType));
            }
            else
            {
              throw new HttpResponseException(new HttpResponseMessage(HttpStatusCode.NotFound));
            }
          } while (query.HasMoreResults);
        }
      }
      if (docs.Count > 0)
        return docs;
      var msg = new StringContent("No documents found for " + resourceType + " Resource");
      var response = new HttpResponseMessage
      {
        StatusCode = HttpStatusCode.NotFound,
        Content = msg
      };
      return response;
    }
  }
}
```

Testing the FhirNotificationApi

The following image demonstrates how swagger was used to to test the FhirNotificationApi.



https://fhirnotificationapi.azurewebsites.net:443/swagger/docs,

FhirNotificationApi

Change Feed suppport provides a sorted list of FHIR documents (Resources) within a DocumentDB collection in the order of which they were modifed Changes in DocumentDB are persisted and can be processed asynchronously, and distributed across one or more consumers for parallel processing

Supports the following scenarios/tasks for incremental processing of changes in DocumentDB collections:

- Read all changes to documents from the beginning, that is, from collection creation.
- Read all changes to future updates to documents from current time.
- Read all changes to documents from a logical version of the collection (ETag). You can checkpoint your consumers based on the returned ETag from incremental read-feed requests.

Use Case

- Implement application-level data tiering and archival, that is, store **hot data** in DocumentDB, and age out **cold data** to Azure Blob Storage or Azure Data Lake Store.

FhirResourceType

/api/FhirResourceType

Get New or Modified FHIR Documents

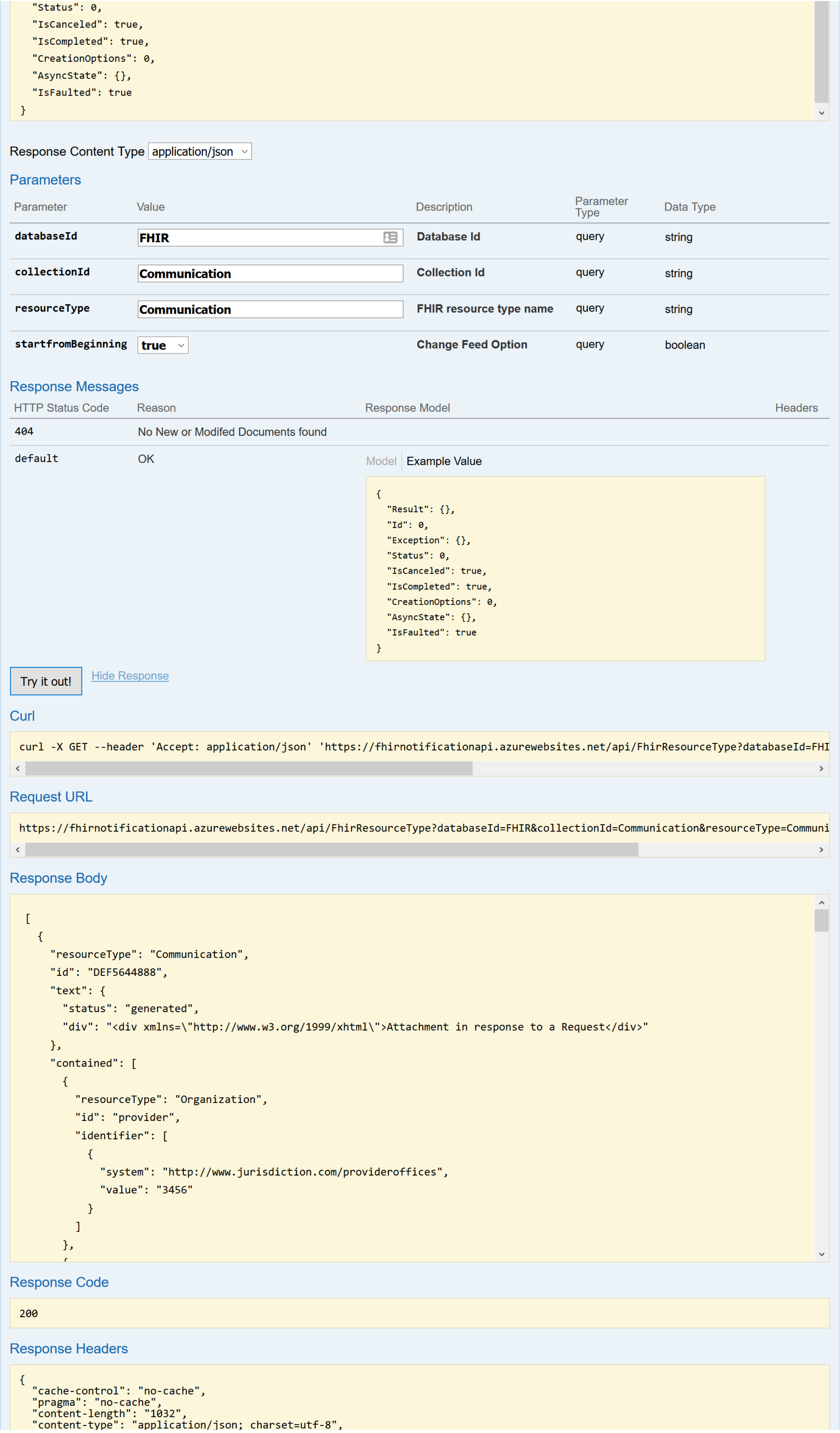
Implementation Notes

Query for new or modifed FHIR Documents By Resource Type from Last Run Date or Begiining of Collection creation

Response Class (Status 200)

OK

```
{
  "Result": {},
  "Id": 0,
  "Exception": {},
```



```
  "Status": 0,
  "IsCanceled": true,
  "IsCompleted": true,
  "CreationOptions": 0,
  "AsyncState": {},
  "IsFaulted": true
}
```

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
`databaseId`	FHIR	Database Id	query	string
`collectionId`	Communication	Collection Id	query	string
`resourceType`	Communication	FHIR resource type name	query	string
`startfromBeginning`	true	Change Feed Option	query	boolean

Response Messages

HTTP Status Code	Reason	Response Model	Headers
404	No New or Modifed Documents found		
default	OK	Model Example Value	

```
{
  "Result": {},
  "Id": 0,
  "Exception": {},
  "Status": 0,
  "IsCanceled": true,
  "IsCompleted": true,
  "CreationOptions": 0,
  "AsyncState": {},
  "IsFaulted": true
}
```

Curl

```
curl -X GET --header 'Accept: application/json' 'https://fhirnotificationapi.azurewebsites.net/api/FhirResourceType?databaseId=FHI
```

Request URL

```
https://fhirnotificationapi.azurewebsites.net/api/FhirResourceType?databaseId=FHIR&collectionId=Communication&resourceType=Communi
```

Response Body

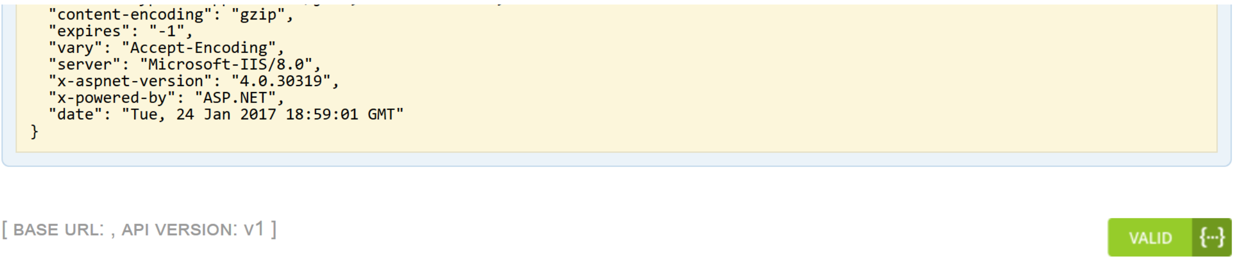
```
[
  {
    "resourceType": "Communication",
    "id": "DEF5644888",
    "text": {
      "status": "generated",
      "div": "<div xmlns=\"http://www.w3.org/1999/xhtml\">Attachment in response to a Request</div>"
    },
    "contained": [
      {
        "resourceType": "Organization",
        "id": "provider",
        "identifier": [
          {
            "system": "http://www.jurisdiction.com/provideroffices",
            "value": "3456"
          }
        ]
      },
```

Response Code

```
200
```

Response Headers

```
{
  "cache-control": "no-cache",
  "pragma": "no-cache",
  "content-length": "1032",
  "content-type": "application/json; charset=utf-8",
```

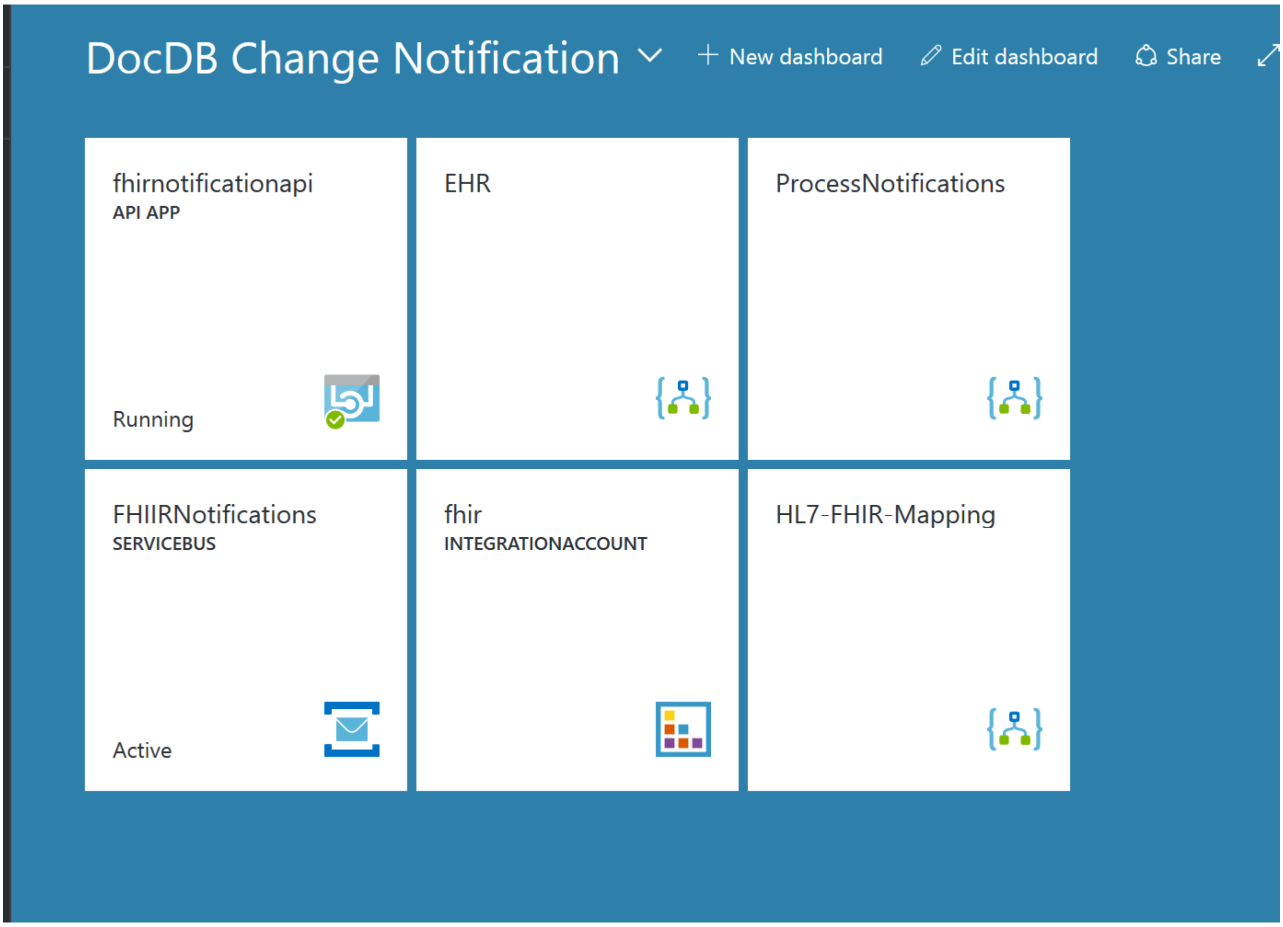


```
    "content-encoding": "gzip",
    "expires": "-1",
    "vary": "Accept-Encoding",
    "server": "Microsoft-IIS/8.0",
    "x-aspnet-version": "4.0.30319",
    "x-powered-by": "ASP.NET",
    "date": "Tue, 24 Jan 2017 18:59:01 GMT"
  }
```

[BASE URL: , API VERSION: v1]

Azure portal dashboard

The following image shows all of the Azure services for this solution running in the Azure portal.



Summary

- You have learned that Azure Cosmos DB has native suppport for notifications for new or modifed documents and how easy it is to use.
- By leveraging Logic Apps, you can create workflows without writing any code.
- Using Azure Service Bus Queues to handle the distribution for the HL7 FHIR documents.

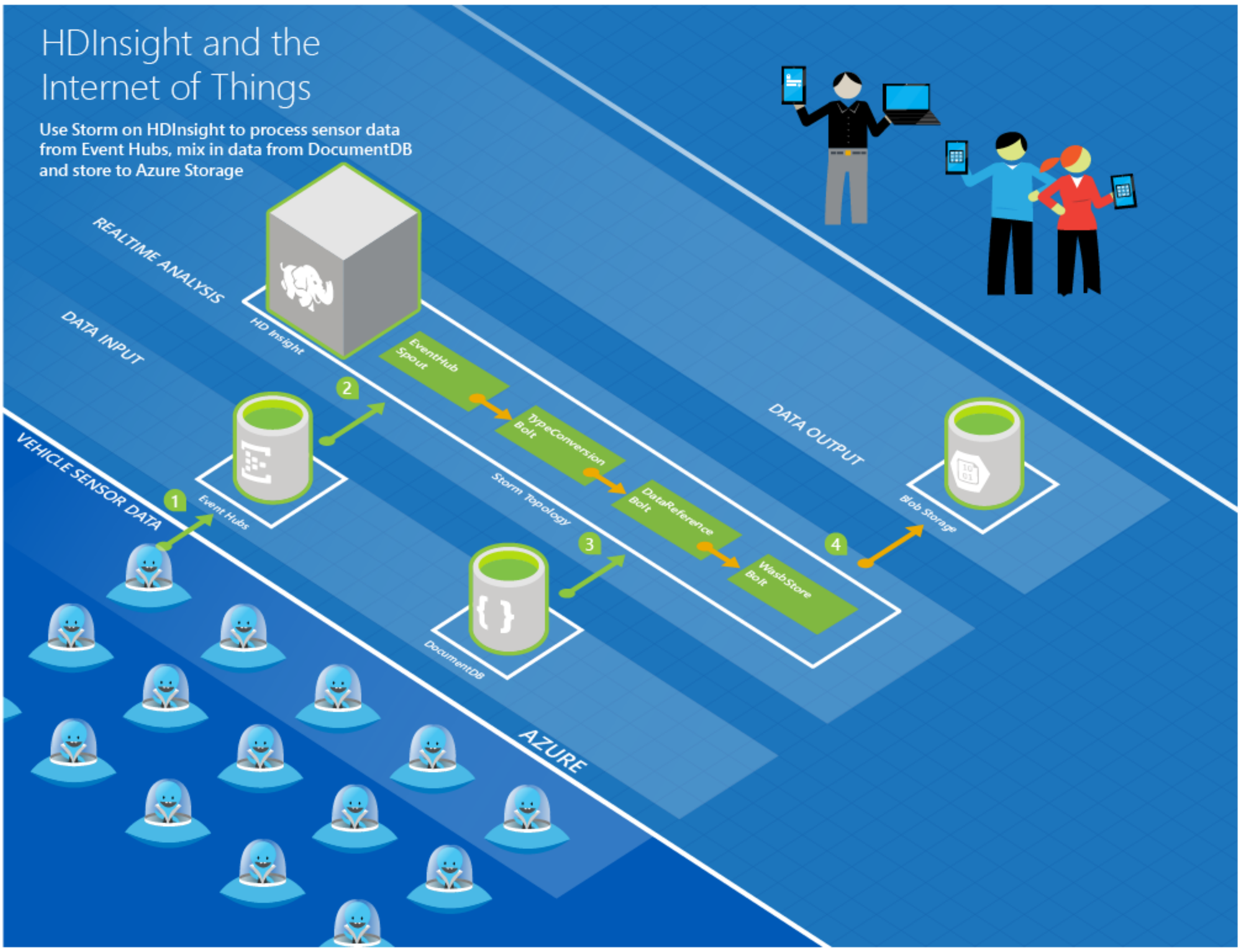
Next steps

For more information about Azure Cosmos DB, see the Azure Cosmos DB home page. For more informaiton about Logic Apps, see Logic Apps.

Process vehicle sensor data from Azure Event Hubs using Apache Storm on HDInsight

5/10/2017 • 1 min to read • Edit Online

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 Cosmos DB. The data is stored into Azure Storage using the Hadoop File System (HDFS).



1 Sensor data (vehicle temperature, speed, ambient temperature, VIN) from vehicles is sent to Azure Event Hubs

2 Data from Event Hubs is processed in real time by a Storm Topology on HDInsight

3 The VIN is used to look up vehicle model information from DocumentDB

4 Processed data is stored in Azure Storage for future use

Microsoft

Overview

Adding sensors to vehicles allows you to predict equipment problems based on historical data trends. It also allows you to make improvements to future versions based on usage pattern 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 is built to handle the massive volume of data generated by sensors. Apache Storm can be used to load and process the data before storing it into HDFS.

Solution

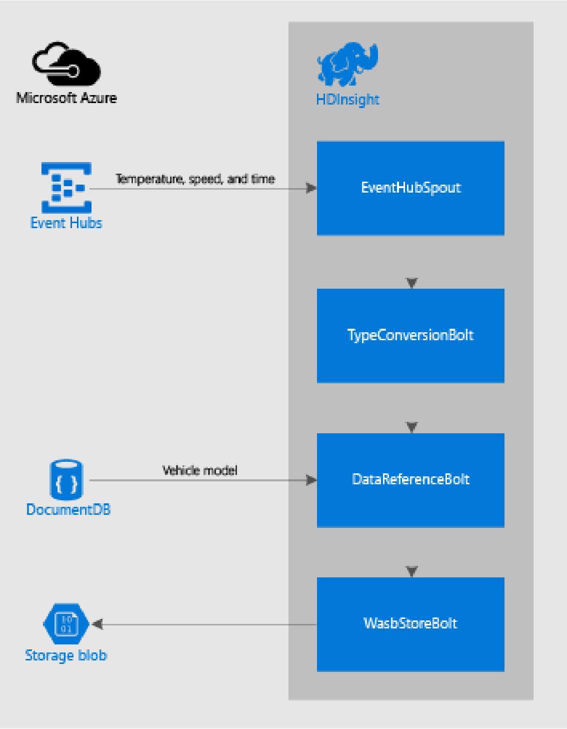
Telemetry data for engine temperature, ambient temperature, and vehicle speed is recorded by sensors. Data is then sent to Event Hubs along with the car's Vehicle Identification Number (VIN) and a time stamp. From there, a Storm Topology running on an Apache Storm on HDInsight cluster reads the data, processes it, and stores it into HDFS.

During processing, the VIN is used to retrieve model information from Cosmos DB. This data 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 following sensor data:
  - Engine temperature
  - Ambient temperature
  - Speed
  - VIN
  - Timestamp
- **DataReferencBolt** - looks up the vehicle model from Cosmos DB using the VIN
- **WasbStoreBolt** - stores the data to HDFS (Azure Storage)

The following image is a diagram of this solution:



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 Azure Cosmos DB: Visualize data using the Power BI connector

5/30/2017 • 9 min to read • Edit Online

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 Cosmos DB account via the Cosmos DB connector for Power BI.

In this Power BI tutorial, we walk through the steps to connect to an Cosmos DB 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 Cosmos DB using Power BI Desktop?
- How can I connect to an Cosmos DB 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 Cosmos DB 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+YhwCH2D2jcczVIR1LNK3nMPNBD31losN7lQ/fkw==
  - Or, to create your own account, see Create an Azure Cosmos DB 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 Azure Cosmos DB data migration tool.

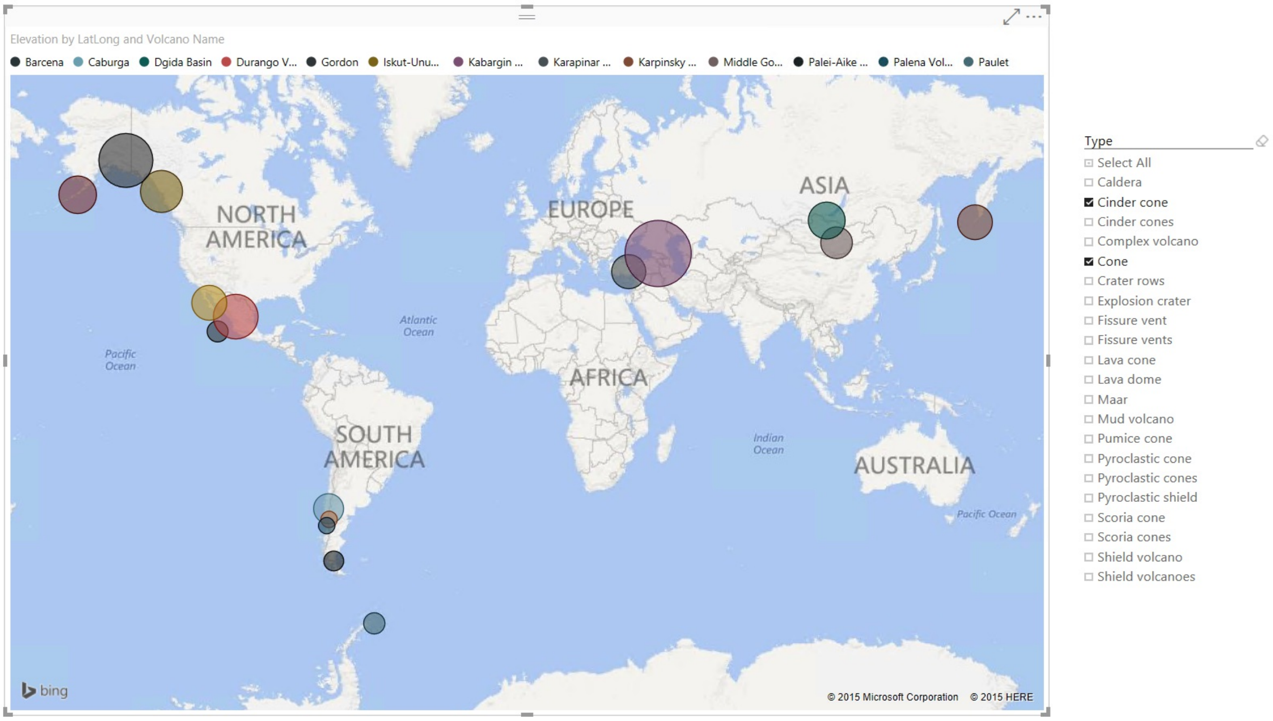
To share your reports in PowerBI.com, you must have an account in PowerBI.com. To learn more about Power BI for Free and Power BI Pro, please visit https://powerbi.microsoft.com/pricing.

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 an Cosmos DB 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"
}
```

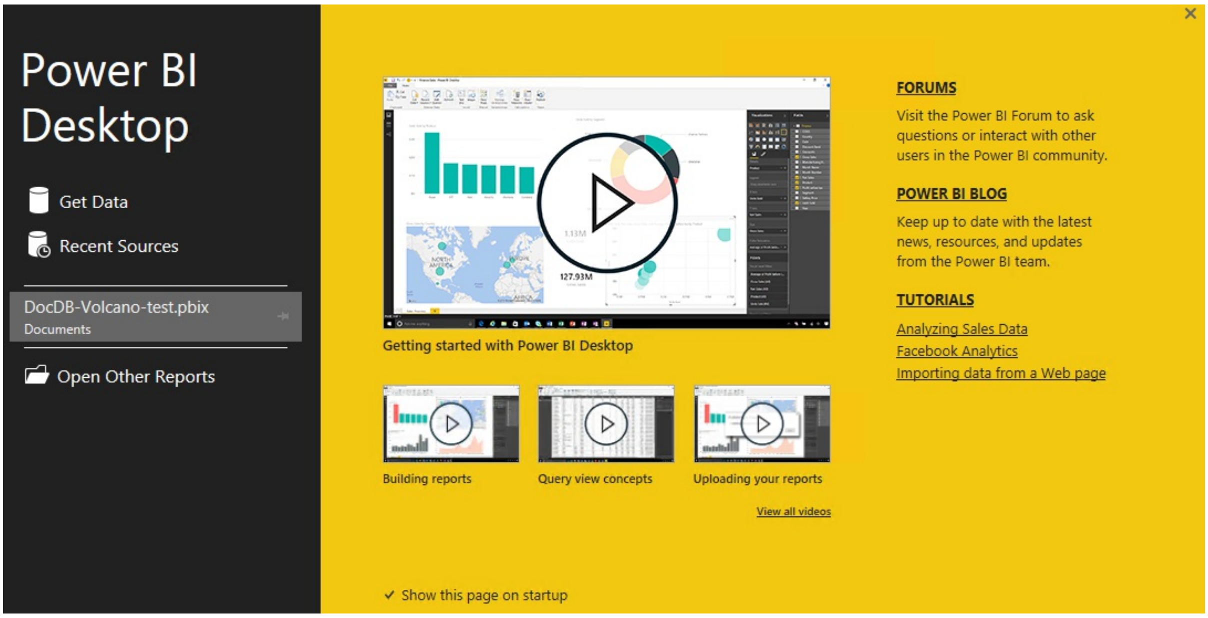
You want to retrieve the volcano data from the Cosmos DB account and visualize data in an interactive Power BI report like the one below.


Elevation by LatLong and Volcano Name
● Barcena ● Caburga ● Dgida Basin ● Durango V... ● Gordon ● Iskut-Unu... ● Kabargin ... ● Karapinar ... ● Karpinsky ... ● Middle Go... ● Palei-Aike ... ● Palena Vol... ● Paulet
NORTH AMERICA
EUROPE
ASIA
AFRICA
SOUTH AMERICA
AUSTRALIA
Atlantic Ocean
Pacific Ocean
Indian Ocean
Pacific Ocean
bing
© 2015 Microsoft Corporation © 2015 HERE

Type
☐ Select All
☐ Caldera
☑ Cinder cone
☐ Cinder cones
☐ Complex volcano
☑ Cone
☐ Crater rows
☐ Explosion crater
☐ Fissure vent
☐ Fissure vents
☐ Lava cone
☐ Lava dome
☐ Maar
☐ Mud volcano
☐ Pumice cone
☐ Pyroclastic cone
☐ Pyroclastic cones
☐ Pyroclastic shield
☐ Scoria cone
☐ Scoria cones
☐ Shield volcano
☐ Shield volcanoes

Ready to give it a try? Let's get started.

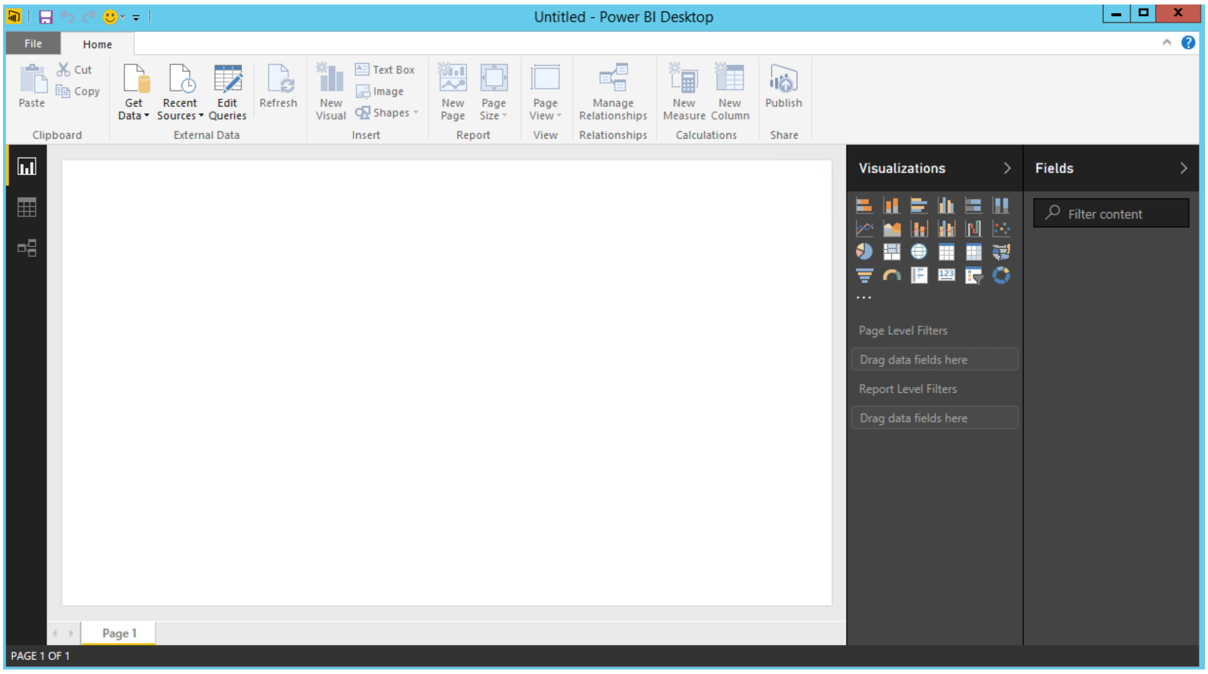
1. Run Power BI Desktop on your workstation.
2. Once Power BI Desktop is launched, a *Welcome* screen is displayed.



Visit the Power BI Forum to ask questions or interact with other users in the Power BI community.

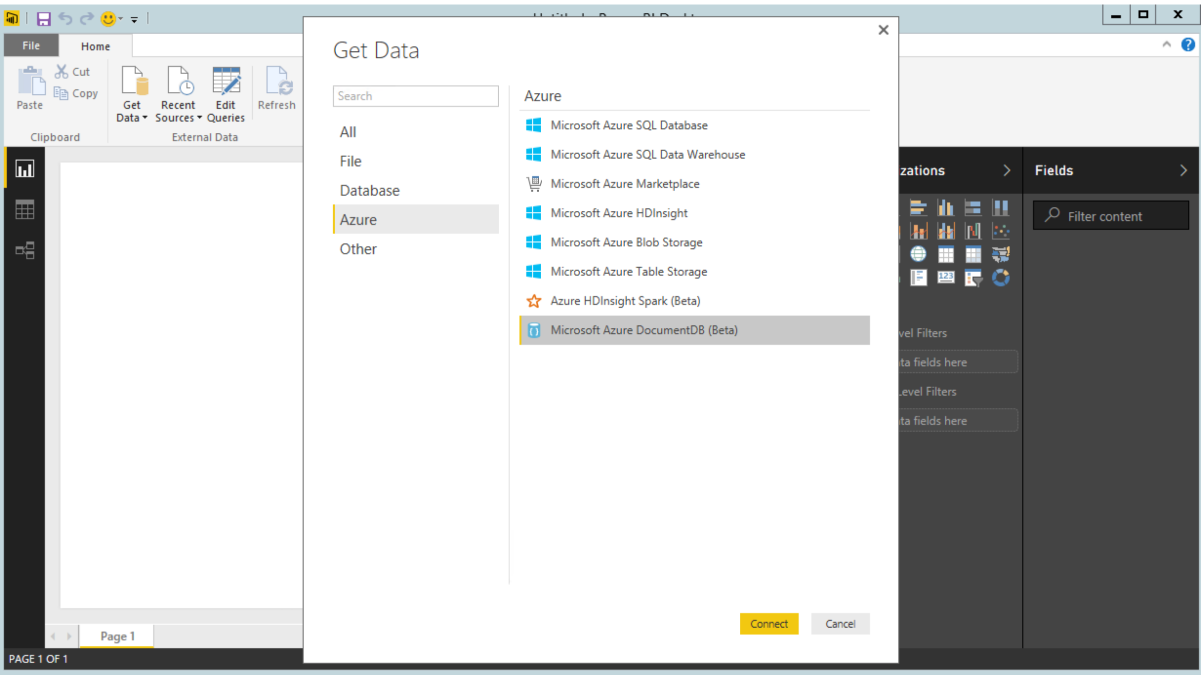
Keep up to date with the latest news, resources, and updates from the Power BI team.

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.



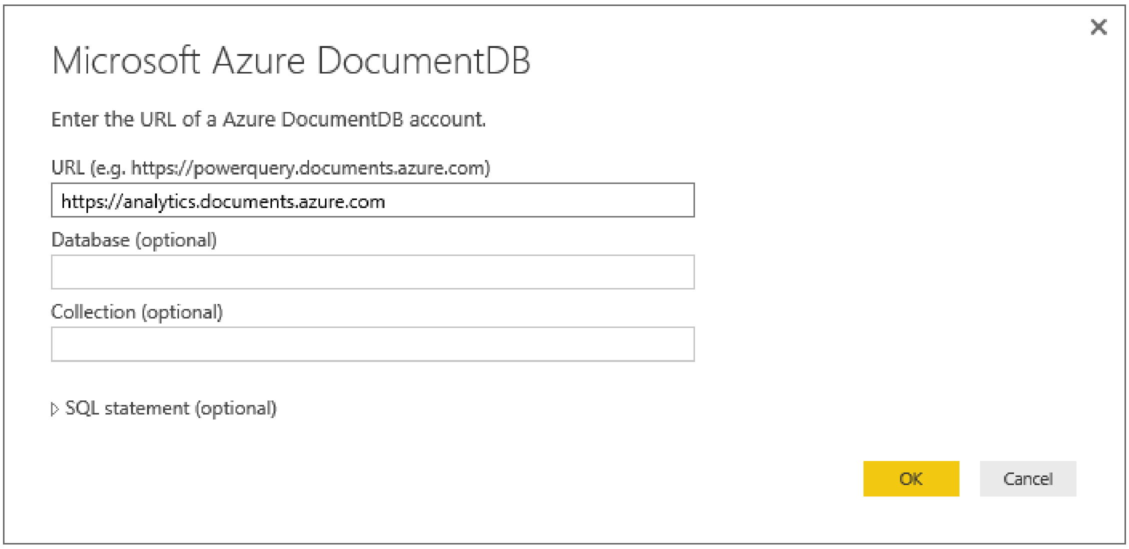
4. Select the **Home** ribbon, then click on **Get Data**. The **Get Data** window should appear.

5. Click on **Azure**, select **Microsoft Azure Cosmos DB (Beta)**, and then click **Connect**. The **Microsoft Azure Cosmos DB Connect** window should appear.



6. Specify the Cosmos DB 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.

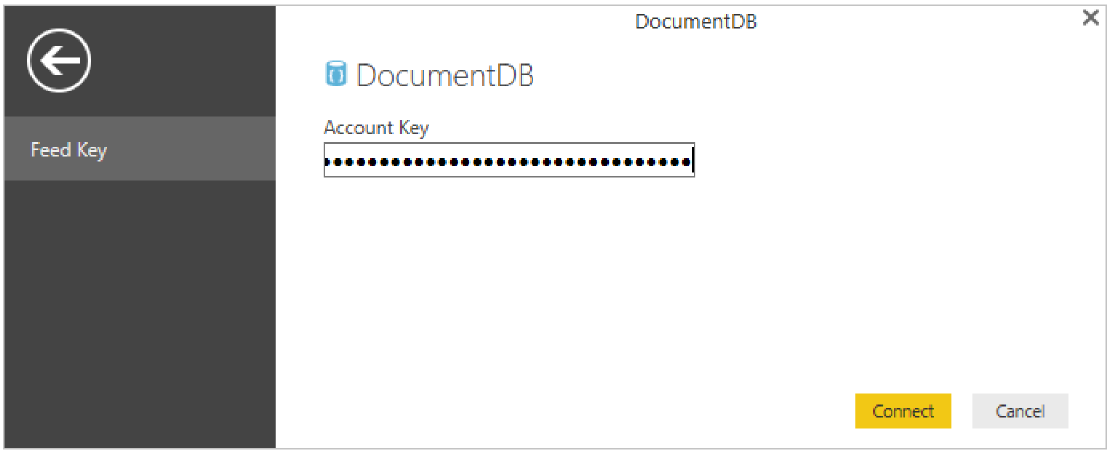
URL (e.g. https://powerquery.documents.azure.com)
https://analytics.documents.azure.com

Database (optional)

Collection (optional)

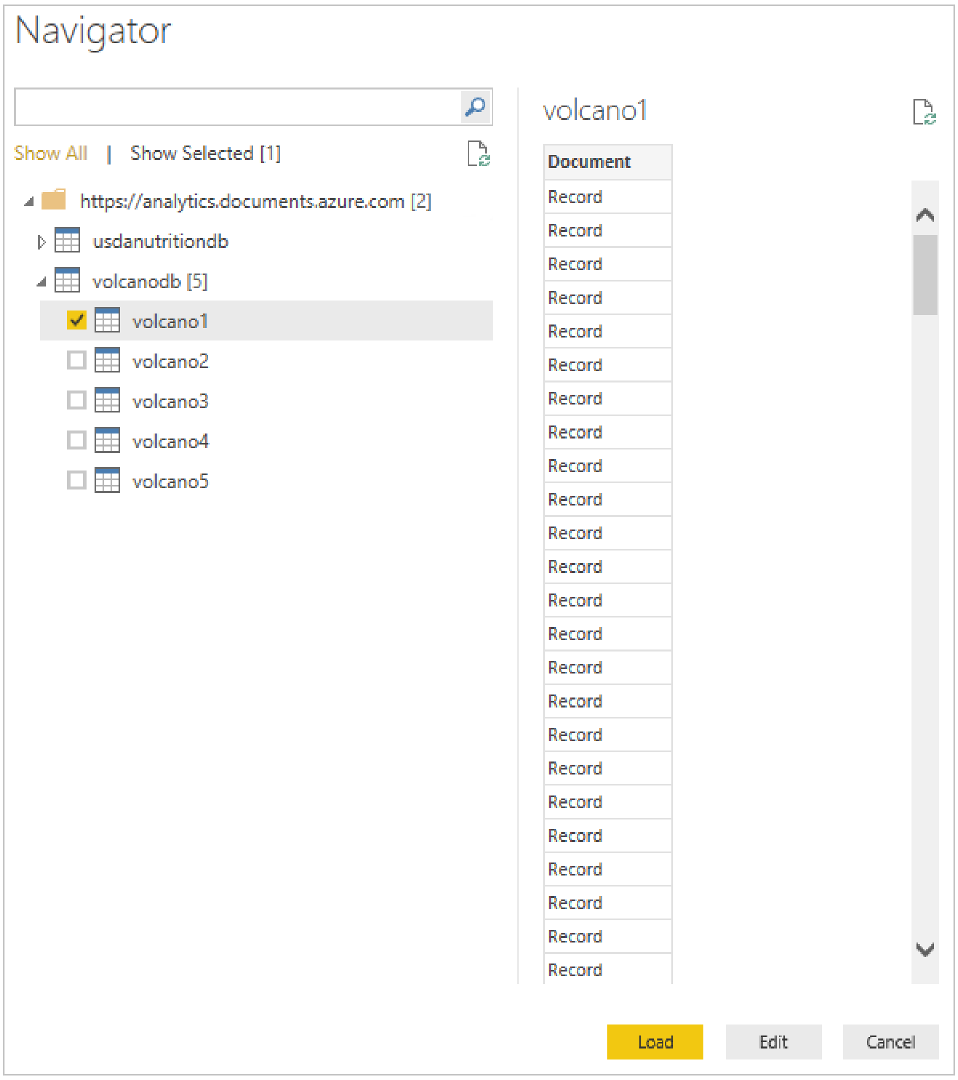
7. If you are connecting to this endpoint for the first time, you will be prompted for the account key. You can retrieve the key from the **Primary Key** box in the **Read-only Keys** blade of the Azure portal, or you can use the demo account, in which case the key is `MSr6kt7Gn0YRQbjd6RbTnTt7VHc5ohaAFu7osF0HdyQmfR+YhwCH2D2jcczVIR1LNK3nMPNBD31losN7lQ/fkw==`. 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**.

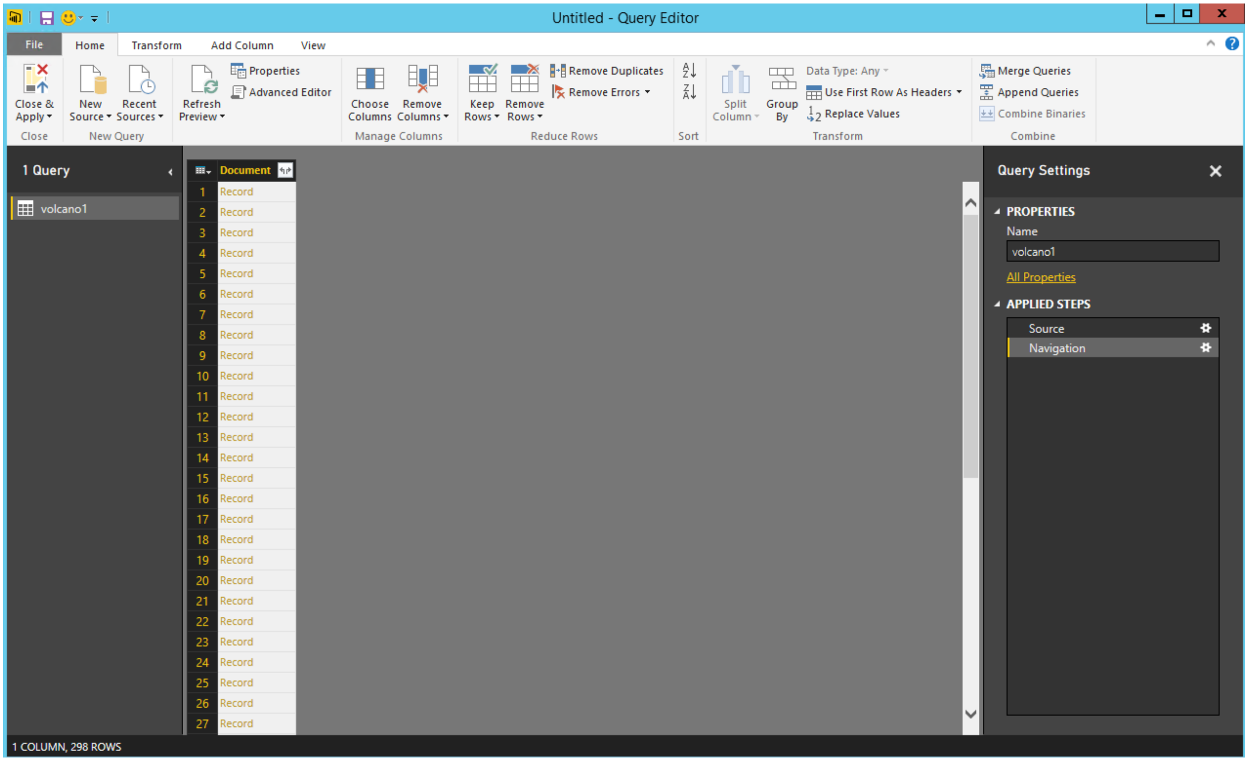
    The Preview pane shows a list of **Record** items. A Document is represented as a **Record** type in Power BI. Similarly, a nested JSON block inside a document is also a **Record**.



11. Click **Edit** to launch the Query Editor so we can transform the data.

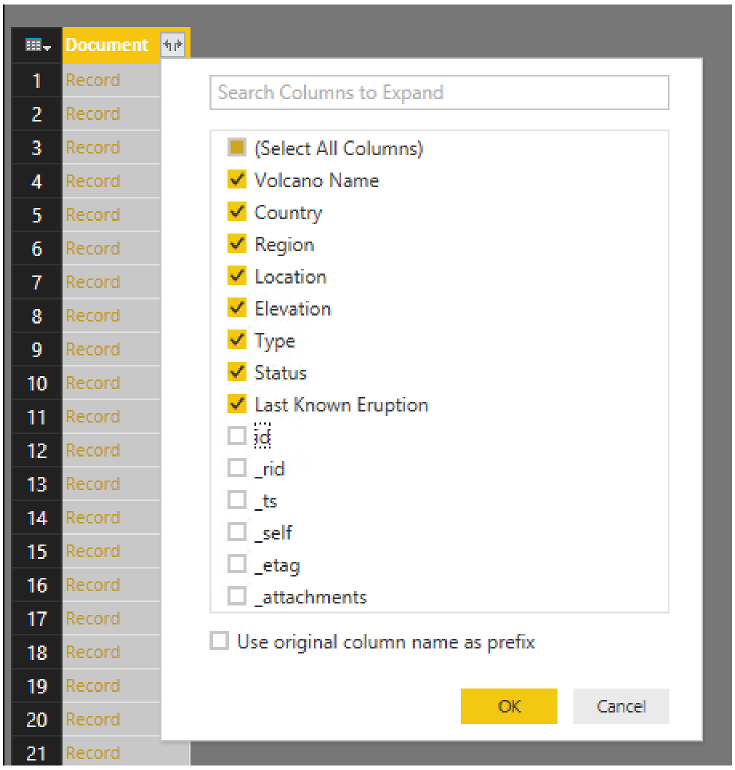
Flattening and transforming JSON documents

1. In the Power BI Query Editor, you should see a **Document** column in the center pane.



	Document
1	Record
2	Record
3	Record
4	Record
5	Record
6	Record
7	Record
8	Record
9	Record
10	Record
11	Record
12	Record
13	Record
14	Record
15	Record
16	Record
17	Record
18	Record
19	Record
20	Record
21	Record
22	Record
23	Record
24	Record
25	Record
26	Record
27	Record

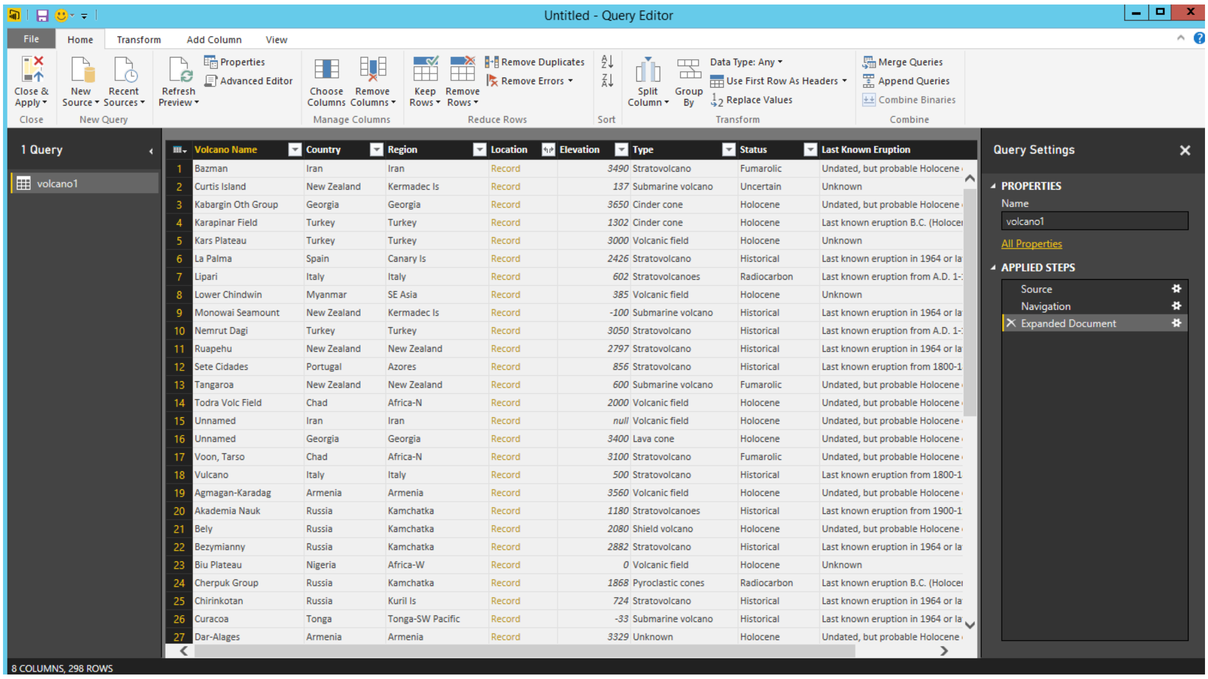
2. Click on the expander at the right side of the **Document** column header. The context menu with a list of fields will appear. Select the fields you need for your report, for instance, Volcano Name, Country, Region, Location, Elevation, Type, Status and Last Know Eruption, and then click **OK**.



■ (Select All Columns)
☑ Volcano Name
☑ Country
☑ Region
☑ Location
☑ Elevation
☑ Type
☑ Status
☑ Last Known Eruption
□ id
□ _rid
□ _ts
□ _self
□ _etag
□ _attachments

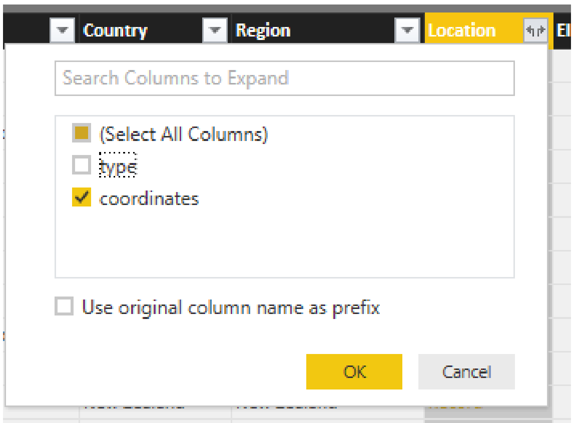
□ Use original column name as prefix

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	Bazman	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. (Holocei
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 Dagi	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	Tangaroa	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	Agmagan-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	Bezymianny	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	Cherpuk Group	Russia	Kamchatka	Record	1868	Pyroclastic cones	Radiocarbon	Last known eruption B.C. (Holocei
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

4. 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.
5. 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**.

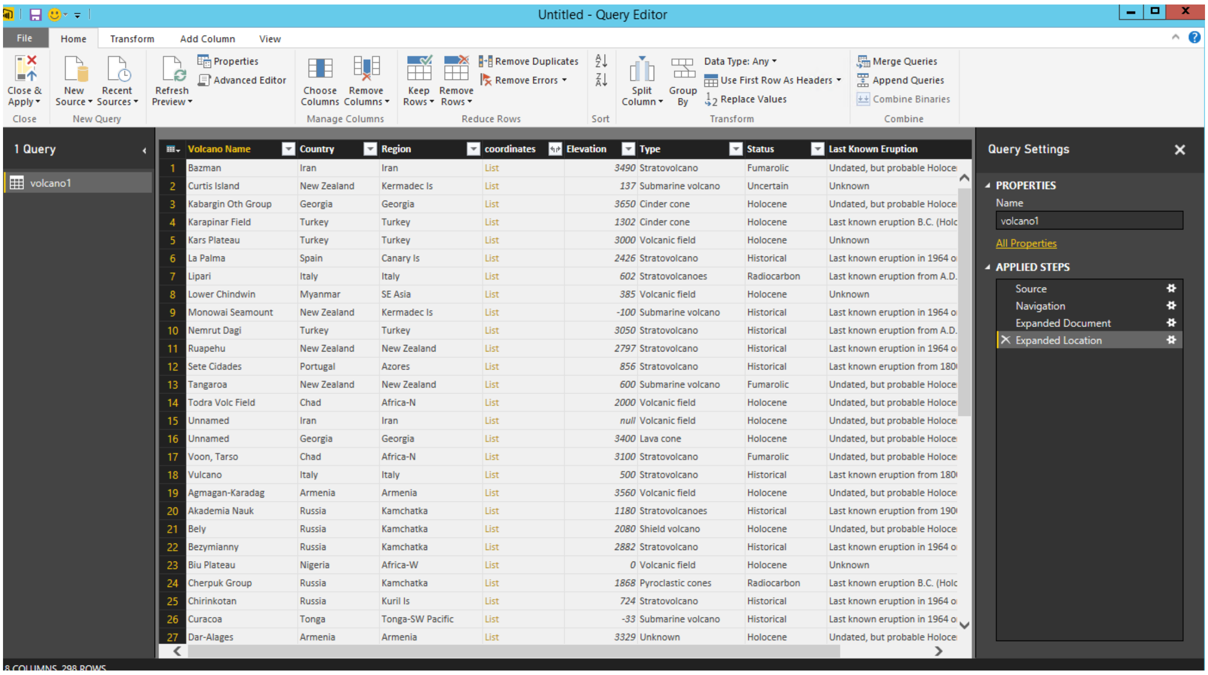


■ (Select All Columns)
□ type
☑ coordinates

□ Use original column name as prefix

6. 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.



Name: volcano1

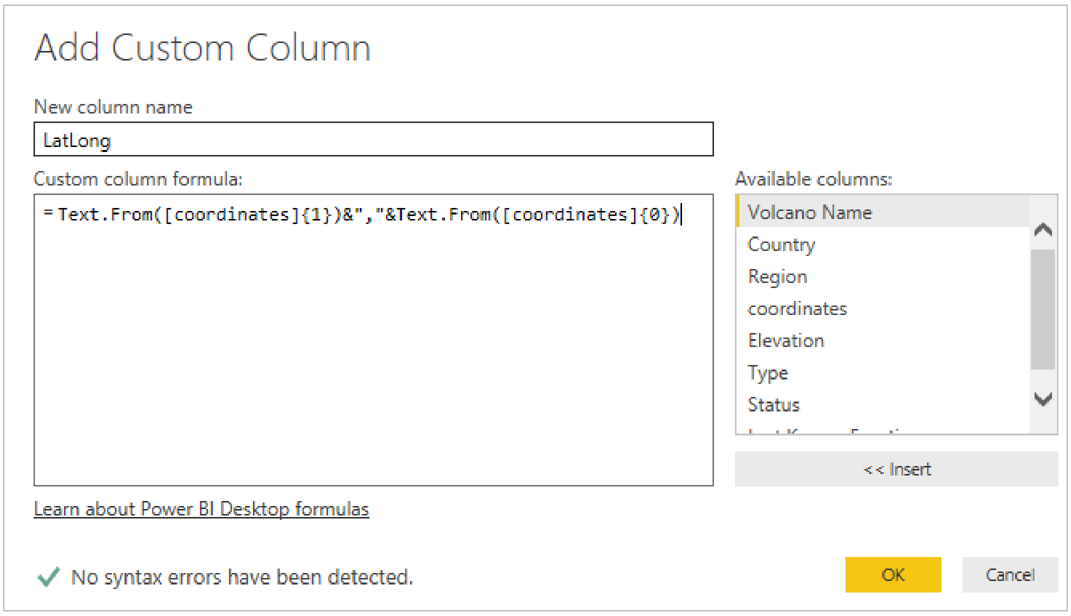
Applied steps: Source, Navigation, Expanded Document, Expanded Location

	Volcano Name	Country	Region	coordinates	Elevation	Type	Status	Last Known Eruption
1	Bazman	Iran	Iran	List	3490	Stratovolcano	Fumarolic	Undated, but probable Holoce
2	Curtis Island	New Zealand	Kermadec Is	List	137	Submarine volcano	Uncertain	Unknown
3	Kabargin Oth Group	Georgia	Georgia	List	3650	Cinder cone	Holocene	Undated, but probable Holoce
4	Karapinar Field	Turkey	Turkey	List	1302	Cinder cone	Holocene	Last known eruption B.C. (Holc
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 Dagi	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 Holoce
14	Todra Volc Field	Chad	Africa-N	List	2000	Volcanic field	Holocene	Undated, but probable Holoce
15	Unnamed	Iran	Iran	List	null	Volcanic field	Holocene	Undated, but probable Holoce
16	Unnamed	Georgia	Georgia	List	3400	Lava cone	Holocene	Undated, but probable Holoce
17	Voon, Tarso	Chad	Africa-N	List	3100	Stratovolcano	Fumarolic	Undated, but probable Holoce
18	Vulcano	Italy	Italy	List	500	Stratovolcano	Historical	Last known eruption from 180
19	Agmagan-Karadag	Armenia	Armenia	List	3560	Volcanic field	Holocene	Undated, but probable Holoce
20	Akademia Nauk	Russia	Kamchatka	List	1180	Stratovolcanoes	Historical	Last known eruption from 190
21	Bely	Russia	Kamchatka	List	2080	Shield volcano	Holocene	Undated, but probable Holoce
22	Bezymianny	Russia	Kamchatka	List	2882	Stratovolcano	Historical	Last known eruption in 1964 o
23	Biu Plateau	Nigeria	Africa-W	List	0	Volcanic field	Holocene	Unknown
24	Cherpuk Group	Russia	Kamchatka	List	1868	Pyroclastic cones	Radiocarbon	Last known eruption B.C. (Holc
25	Chirinkotan	Russia	Kuril Is	List	724	Stratovolcano	Historical	Last known eruption in 1964 o
26	Curacoa	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 Holoce

8 COLUMNS, 298 ROWS

7. 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.
8. Provide a name for the new column, e.g. LatLong.
9. 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.



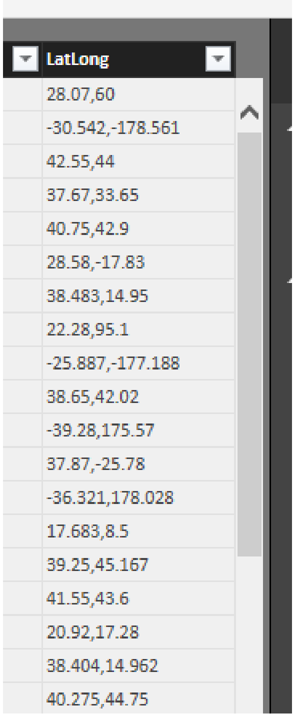
New column name: LatLong

Custom column formula:

```
= Text.From([coordinates]{1})&","&Text.From([coordinates]{0})
```

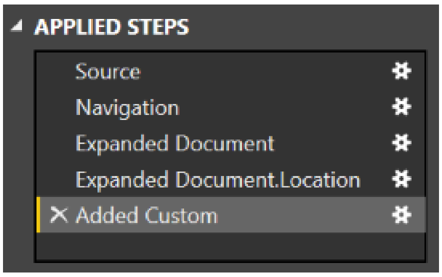
No syntax errors have been detected.

10. 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:

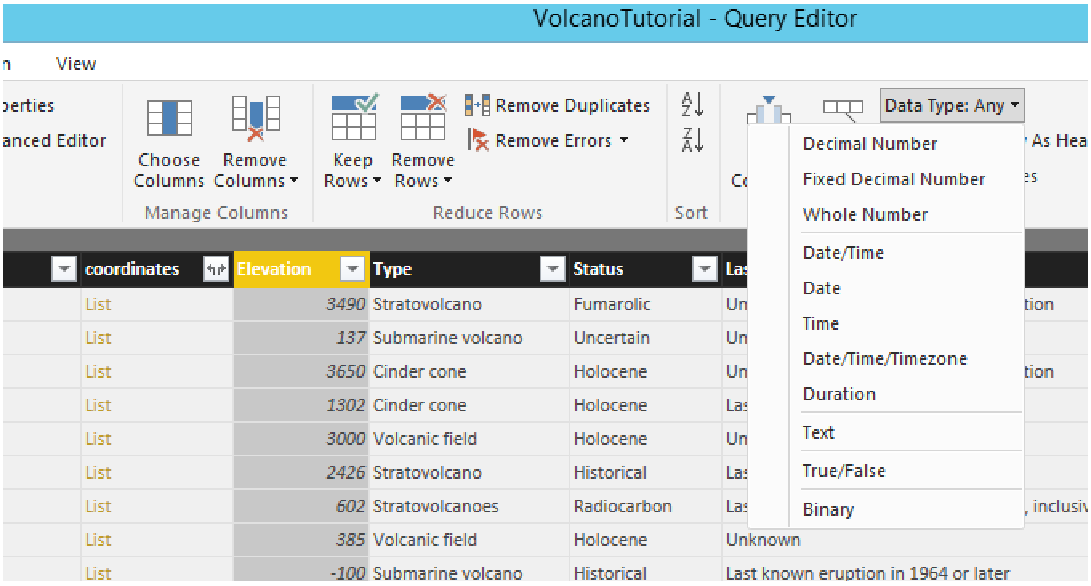


APPLIED STEPS

Source
Navigation
Expanded Document
Expanded Document.Location
Added Custom

If your steps are different, delete the extra steps and try adding the custom column again.

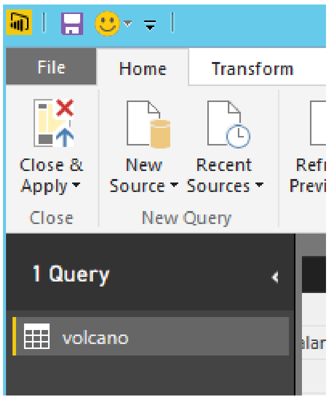
11. We have now completed flattening the data into tabular format. You can leverage all of the features available in the Query Editor to shape and transform data in Cosmos DB. If you're using the sample, change the data type for Elevation to **Whole number** by changing the **Data Type** on the **Home** ribbon.



VolcanoTutorial - Query Editor

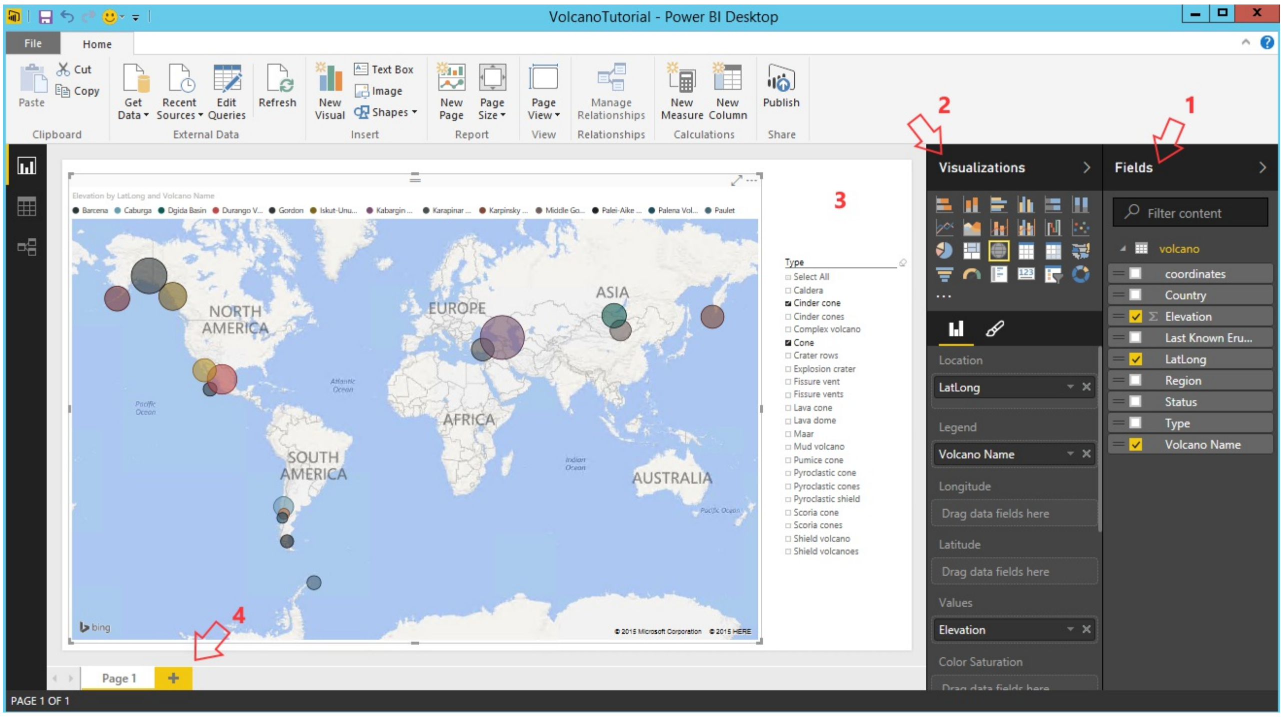
coordinates	Elevation	Type	Status	Las	
List	3490	Stratovolcano	Fumarolic	Un	tion
List	137	Submarine volcano	Uncertain	Un	
List	3650	Cinder cone	Holocene	Un	tion
List	1302	Cinder cone	Holocene	Las	
List	3000	Volcanic field	Holocene	Un	
List	2426	Stratovolcano	Historical	Las	
List	602	Stratovolcanoes	Radiocarbon	Las	, inclusiv
List	385	Volcanic field	Holocene	Unknown	
List	-100	Submarine volcano	Historical	Last known eruption in 1964 or later	

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.



Location: LatLong

Legend: Volcano Name

Longitude: Drag data fields here

Latitude: Drag data fields here

Values: Elevation

Color Saturation

In the Report view, you should find:

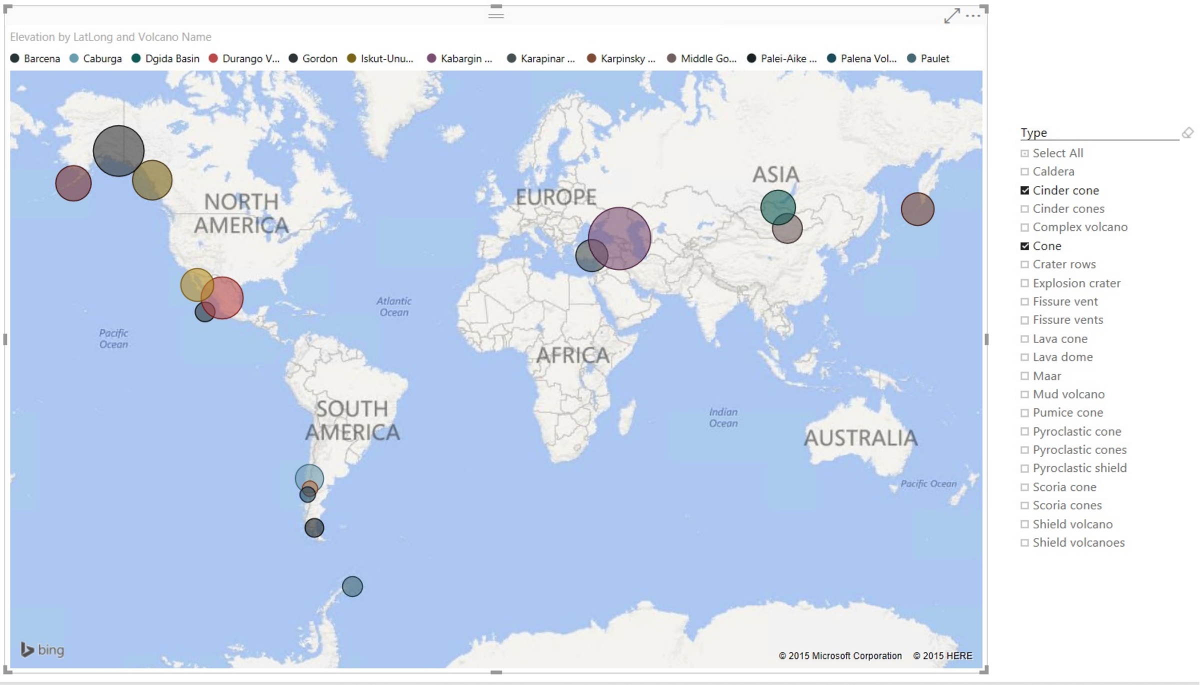
1. The **Fields** pane, this is where you will see a list of data models with fields you can use for your reports.
2. The **Visualizations** pane. A report can contain a single or multiple visualizations. Pick the visual types fitting your needs from the **Visualizations** pane.
3. The **Report** canvas, 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.



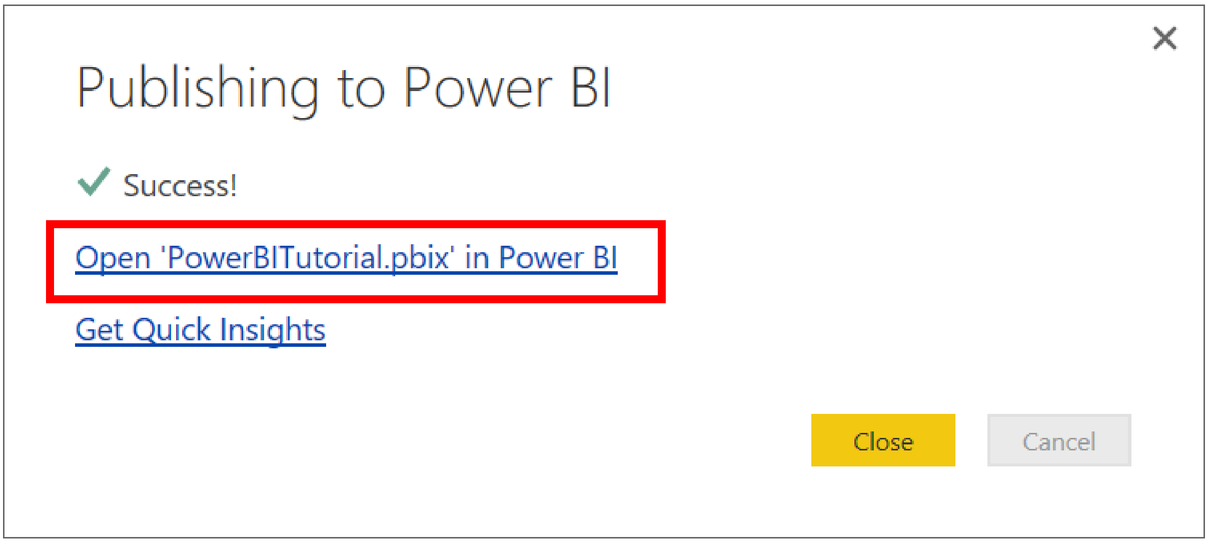
Elevation by LatLong and Volcano Name
● Barcena ● Caburga ● Dgida Basin ● Durango V... ● Gordon ● Iskut-Unu... ● Kabargin ... ● Karapinar ... ● Karpinsky ... ● Middle Go... ● Palei-Aike ... ● Palena Vol... ● Paulet
NORTH AMERICA
EUROPE
ASIA
AFRICA
SOUTH AMERICA
AUSTRALIA
Atlantic Ocean
Pacific Ocean
Indian Ocean
Pacific Ocean
bing
© 2015 Microsoft Corporation © 2015 HERE

Type
- □ Select All
- □ Caldera
- ☑ Cinder cone
- □ Cinder cones
- □ Complex volcano
- ☑ Cone
- □ Crater rows
- □ Explosion crater
- □ Fissure vent
- □ Fissure vents
- □ Lava cone
- □ Lava dome
- □ Maar
- □ Mud volcano
- □ Pumice cone
- □ Pyroclastic cone
- □ Pyroclastic cones
- □ Pyroclastic shield
- □ Scoria cone
- □ Scoria cones
- □ Shield volcano
- □ Shield volcanoes

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.



Publishing to Power BI

✓ Success!

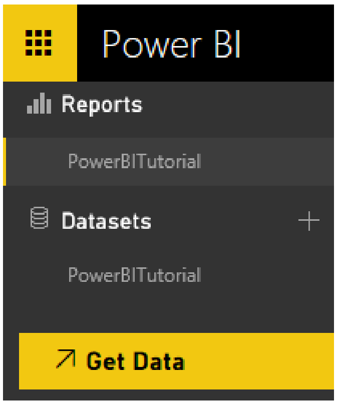
Open 'PowerBITutorial.pbix' in Power BI

Get Quick Insights

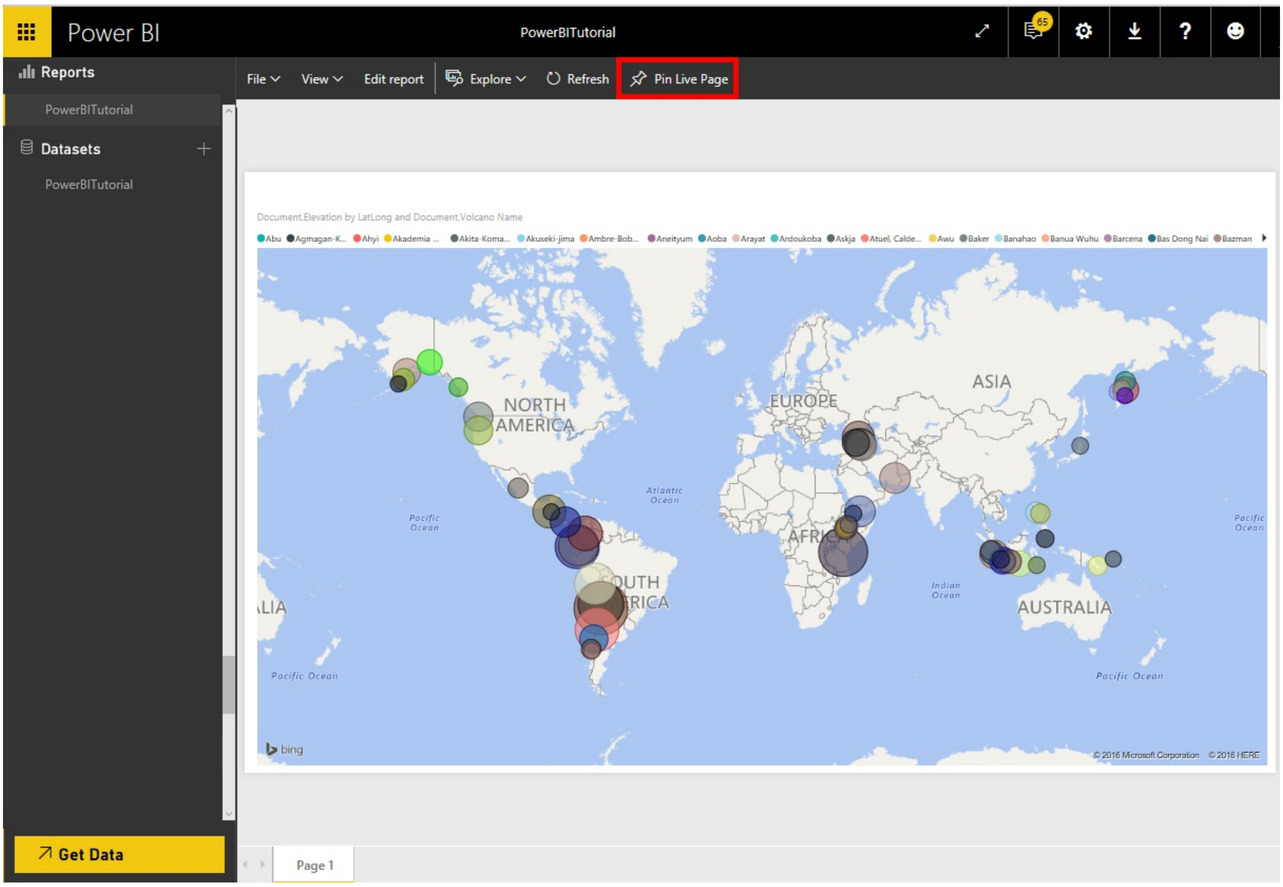
Create a dashboard in PowerBI.com

Now that you have a report, lets 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.



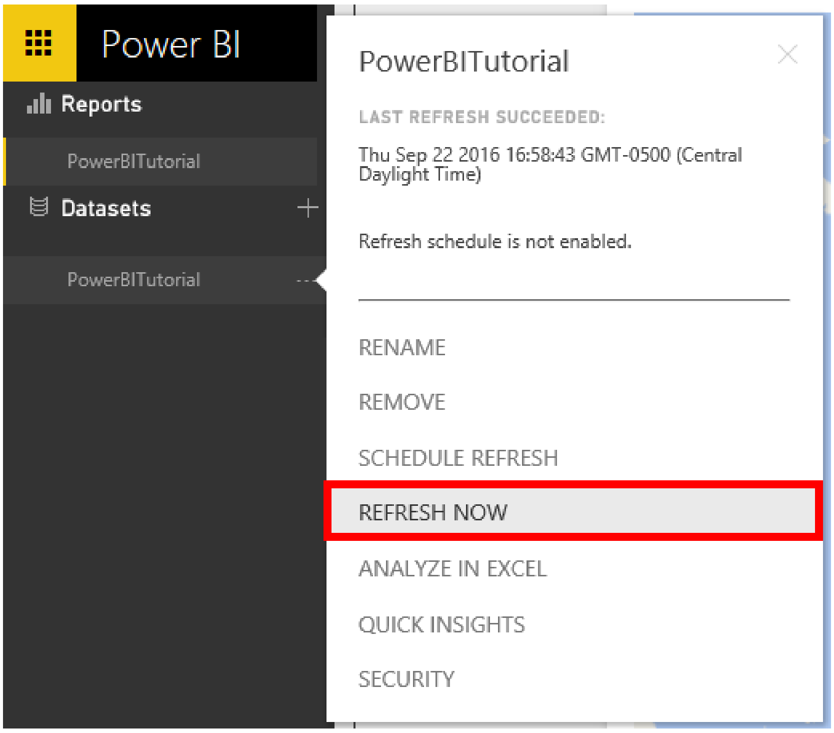
Then follow the instructions in Pin a tile from a report to create a new dashboard.

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.



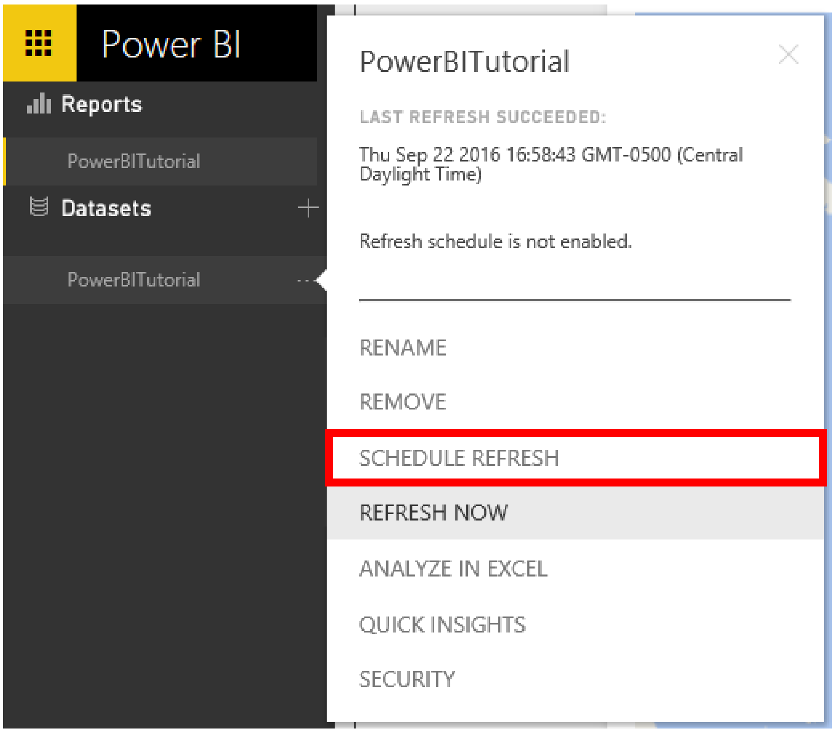
LAST REFRESH SUCCEEDED:

Thu Sep 22 2016 16:58:43 GMT-0500 (Central Daylight Time)

Refresh schedule is not enabled.

For a scheduled refresh, do the following.

1. Click **Schedule Refresh** in the action list.



LAST REFRESH SUCCEEDED:

Thu Sep 22 2016 16:58:43 GMT-0500 (Central Daylight Time)

Refresh schedule is not enabled.

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 Cosmos DB 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 Cosmos DB, see the Azure Cosmos DB documentation landing page.

Connect to Azure Cosmos DB using BI analytics tools with the ODBC driver

5/30/2017 • 11 min to read • Edit Online

The Azure Cosmos DB ODBC driver enables you to connect to Azure Cosmos DB using BI analytics tools such as SQL Server Integration Services, Power BI Desktop, and Tableau so that you can analyze and create visualizations of your Azure Cosmos DB data in those solutions.

The Azure Cosmos DB ODBC driver is ODBC 3.8 compliant and supports ANSI SQL-92 syntax. The driver offers rich features to help you renormalize data in Azure Cosmos DB. Using the driver, you can represent data in Azure Cosmos DB as tables and views. The driver enables you to perform SQL operations against the tables and views including group by queries, inserts, updates, and deletes.

Why do I need to normalize my data?

Azure Cosmos DB is a schemaless database, so it enables rapid development of apps by enabling applications to iterate their data model on the fly and not confine them to a strict schema. A single Azure Cosmos DB database can contain JSON documents of various structures. This is great for rapid application development, but when you want to analyze and create reports of your data using data analytics and BI tools, the data often needs to be flattened and adhere to a specific schema.

This is where the ODBC driver comes in. By using the ODBC driver, you can now renormalized data in Azure Cosmos DB into tables and views fitting to your data analytic and reporting needs. The renormalized schemas have no impact on the underlying data and do not confine developers to adhere to them, they simply enable you to leverage ODBC-compliant tools to access the data. So now your Azure Cosmos DB database will not only be a favorite for your development team, but your data analysts will love it too.

Now lets get started with the ODBC driver.

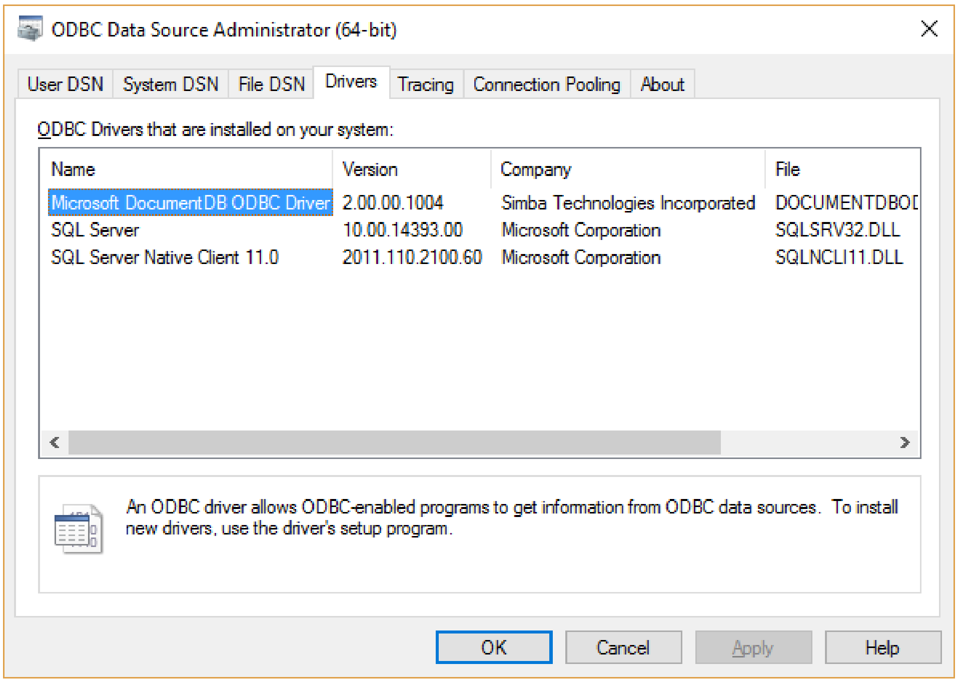
Step 1: Install the Azure Cosmos DB ODBC driver

1. Download the drivers for your environment:

   - Microsoft Azure Cosmos DB ODBC 64-bit.msi for 64-bit Windows
   - Microsoft Azure Cosmos DB ODBC 32x64-bit.msi for 32-bit on 64-bit Windows
   - Microsoft Azure Cosmos DB ODBC 32-bit.msi for 32-bit Windows

   Run the msi file locally, which starts the **Microsoft Azure Cosmos DB ODBC Driver Installation Wizard**.

2. Complete the installation wizard using the default input to install the ODBC driver.
3. Open the **ODBC Data source Administrator** app on your computer, you can do this by typing **ODBC Data sources** in the Windows search box. You can confirm the driver was installed by clicking the **Drivers** tab and ensuring **Microsoft Azure Cosmos DB ODBC Driver** is listed.



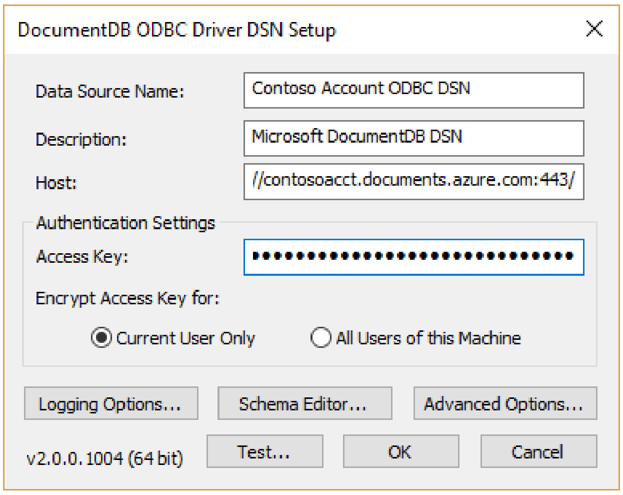
ODBC Drivers that are installed on your system:

Name	Version	Company	File
Microsoft DocumentDB ODBC Driver	2.00.00.1004	Simba Technologies Incorporated	DOCUMENTDBOI
SQL Server	10.00.14393.00	Microsoft Corporation	SQLSRV32.DLL
SQL Server Native Client 11.0	2011.110.2100.60	Microsoft Corporation	SQLNCLI11.DLL

An ODBC driver allows ODBC-enabled programs to get information from ODBC data sources. To install new drivers, use the driver's setup program.

Step 2: Connect to your Azure Cosmos DB database

1. After Installing the Azure Cosmos DB ODBC driver, in the **ODBC Data Source Administrator** window, click **Add**. You can create a User or System DSN. In this example, we are creating a User DSN.
2. In the **Create New Data Source** window, select **Microsoft Azure Cosmos DB ODBC Driver**, and then click **Finish**.
3. In the **Azure Cosmos DB ODBC Driver SDN Setup** window, fill in the following:



Data Source Name: Contoso Account ODBC DSN

Description: Microsoft DocumentDB DSN

Host: //contosoacct.documents.azure.com:443/

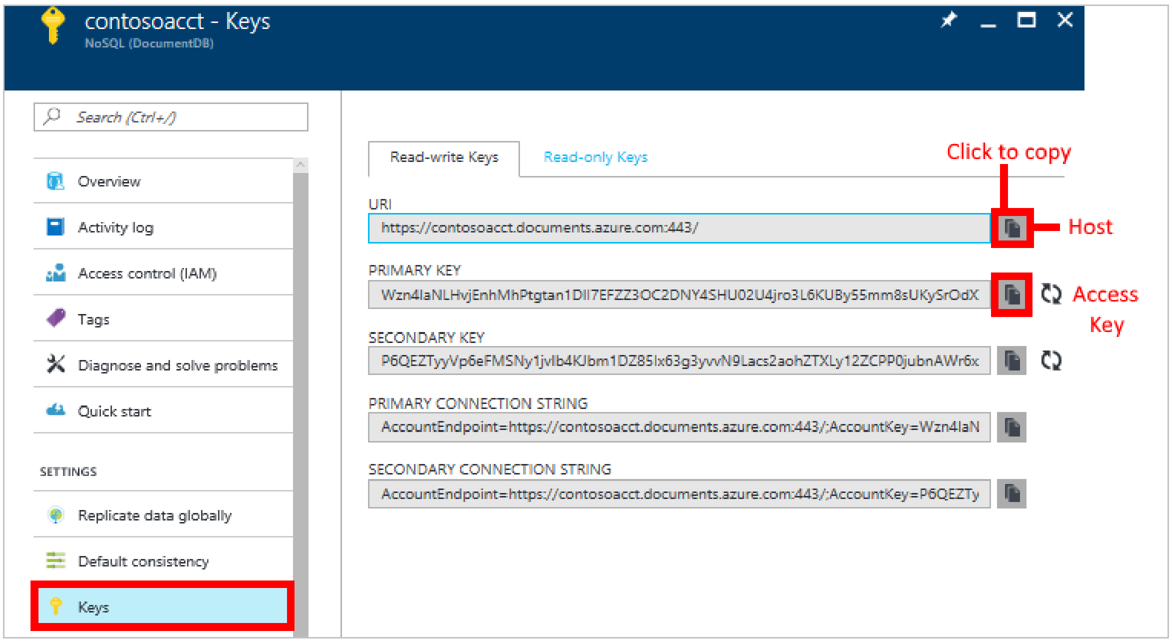
Authentication Settings

Access Key: ••••••••••••••••••••••••••••

Encrypt Access Key for:

● Current User Only ○ All Users of this Machine

- **Data Source Name**: Your own friendly name for the ODBC DSN. This name is unique to your Azure Cosmos DB account, so name it appropriately if you have multiple accounts.
- **Description**: A brief description of the data source.
- **Host**: URI for your Azure Cosmos DB account. You can retrieve this from the Azure Cosmos DB Keys blade in the Azure portal, as shown in the following screenshot.
- **Access Key**: The primary or secondary, read-write or read-only key from the Azure Cosmos DB Keys blade in the Azure portal as shown in the following screenshot. We recommend you use the read-only key if the DSN is used for read-only data processing and reporting.



Click to copy

URI: https://contosoacct.documents.azure.com:443/

Host

PRIMARY KEY: Wzn4laNLHvjEnhMhPtgtan1DII7EFZZ3OC2DNY4SHU02U4jro3L6KUBy55mm8sUKySrOdX

Access Key

SECONDARY KEY: P6QEZTyyVp6eFMSNy1jvIb4KJbm1DZ85Ix63g3yvvN9Lacs2aohZTXLy12ZCPP0jubnAWr6x

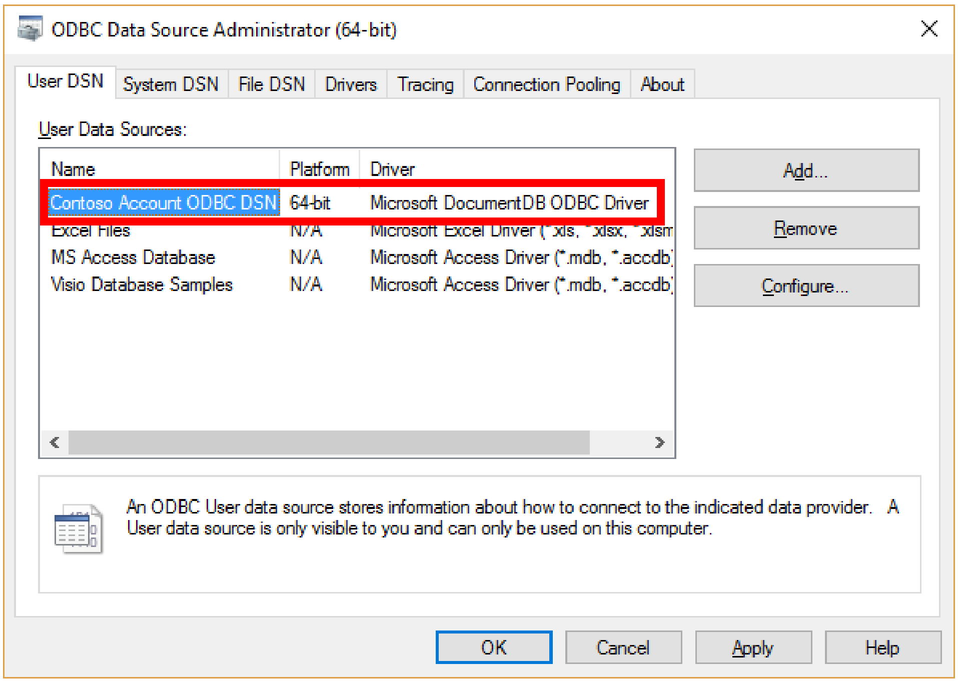
PRIMARY CONNECTION STRING: AccountEndpoint=https://contosoacct.documents.azure.com:443/;AccountKey=Wzn4laN

SECONDARY CONNECTION STRING: AccountEndpoint=https://contosoacct.documents.azure.com:443/;AccountKey=P6QEZTy

- **Encrypt Access Key for**: Select the best choice based on the users of this machine.

4. Click the **Test** button to make sure you can connect to your Azure Cosmos DB account.
5. Click **Advanced Options** and set the following values:

   - **Query Consistency**: Select the consistency level for your operations. The default is Session.
   - **Number of Retries**: Enter the number of times to retry an operation if the initial request does not complete due to service throttling.
   - **Schema File**: You have a number of options here.
     - By default, leaving this entry as is (blank), the driver scans the first page data for all collections to determine the schema of each collection. This is known as Collection Mapping. Without a schema file defined, the driver has to perform the scan for each driver session and could result in a higher start up time of an application using the DSN. We recommend that you always associate a schema file for a DSN.
     - If you already have a schema file (possibly one that you created using the Schema Editor), you can click **Browse**, navigate to your file, click **Save**, and then click **OK**.
     - If you want to create a new schema, click **OK**, and then click **Schema Editor** in the main window. Then proceed to the Schema Editor information. Upon creating the new schema file, please remember to go back to the **Advanced Options** window to include the newly created schema file.
6. Once you complete and close the **Azure Cosmos DB ODBC Driver DSN Setup** window, the new User DSN is added to the User DSN tab.



User Data Sources:

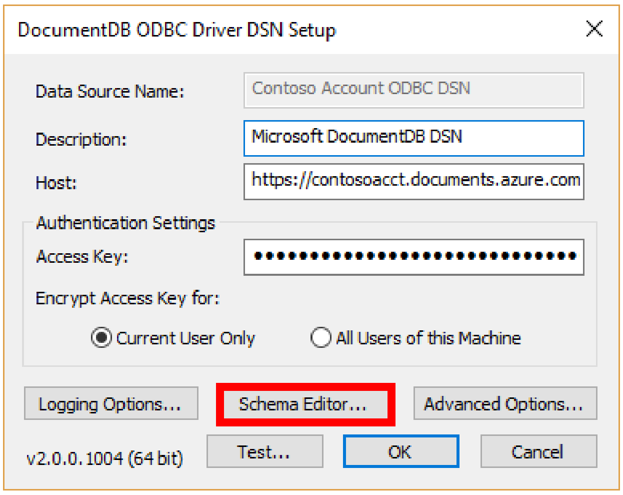
Name	Platform	Driver
Contoso Account ODBC DSN	64-bit	Microsoft DocumentDB ODBC Driver
Excel Files	N/A	Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm
MS Access Database	N/A	Microsoft Access Driver (*.mdb, *.accdb)
Visio Database Samples	N/A	Microsoft Access Driver (*.mdb, *.accdb)

An ODBC User data source stores information about how to connect to the indicated data provider. A User data source is only visible to you and can only be used on this computer.

Step 3: Create a schema definition using the collection mapping method

There are two types of sampling methods that you can use: **collection mapping** or **table-delimiters**. A sampling session can utilize both sampling methods, but each collection can only use a specific sampling method. The steps below create a schema for the data in one or more collections using the collection mapping method. This sampling method retrieves the data in the page of a collection to determine the structure of the data. It transposes a collection to a table on the ODBC side. This sampling method is efficient and fast when the data in a collection is homogenous. If a collection contains heterogenous type of data, we recommend you use the table-delimiters mapping method as it provides a more robust sampling method to determine the data structures in the collection.

1. After completing steps 1-4 in Connect to your Azure Cosmos DB database, click **Schema Editor** in the **Azure Cosmos DB ODBC Driver DSN Setup** window.



Data Source Name: Contoso Account ODBC DSN

Description: Microsoft DocumentDB DSN

Host: https://contosoacct.documents.azure.com

Authentication Settings

Access Key: ••••••••••••••••••••••••••••••

Encrypt Access Key for:

● Current User Only ○ All Users of this Machine

v2.0.0.1004 (64 bit)

2. In the **Schema Editor** window, click **Create New**. The **Generate Schema** window displays all the collections in the Azure Cosmos DB account.
3. Select one or more collections to sample, and then click **Sample**.
4. In the **Design View** tab, the database, schema, and table are represented. In the table view, the scan displays the set of properties associated with the column names (SQL Name, Source Name, etc.). For each column, you can modify the column SQL name, the SQL type, SQL length (if applicable), Scale (if applicable), Precision (if applicable) and Nullable.

- You can set **Hide Column** to **true** if you want to exclude that column from query results. Columns marked Hide Column = true are not returned for selection and projection, although they are still part of the schema. For example, you can hide all of the Azure Cosmos DB system required properties starting with "_".
- The **id** column is the only field that cannot be hidden as it is used as the primary key in the normalized schema.

5. Once you have finished defining the schema, click **File** | **Save**, navigate to the directory to save the schema, and then click **Save**.

   If in the future you want to use this schema with a DSN, open the Azure Cosmos DB ODBC Driver DSN Setup window (via the ODBC Data Source Administrator), click Advanced Options, and then in the Schema File box, navigate to the saved schema. Saving a schema file to an existing DSN modifies the DSN connection to scope to the data and structure defined by schema.

Step 4: Create a schema definition using the table-delimiters mapping method

There are two types of sampling methods that you can use: **collection mapping** or **table-delimiters**. A sampling session can utilize both sampling methods, but each collection can only use a specific sampling method.

The following steps create a schema for the data in one or more collections using the **table-delimiters** mapping method. We recommend that you use this sampling method when your collections contain heterogeneous type of data. You can use this method to scope the sampling to a set of attributes and its corresponding values. For example, if a document contains a "Type" property, you can scope the sampling to the values of this property. The end result of the sampling would be a set of tables for each of the values for Type you have specified. For example, Type = Car will produce a Car table while Type = Plane would produce a Plane table.

1. After completing steps 1-4 in Connect to your Azure Cosmos DB database, click **Schema Editor** in the Azure Cosmos DB ODBC Driver DSN Setup window.
2. In the **Schema Editor** window, click **Create New**. The **Generate Schema** window displays all the collections in the Azure Cosmos DB account.
3. Select a collection on the **Sample View** tab, in the **Mapping Definition** column for the collection, click **Edit**. Then in the **Mapping Definition** window, select **Table Delimiters** method. Then do the following:

   a. In the **Attributes** box, type the name of a delimiter property. This is a property in your document that you want to scope the sampling to, for instance, City and press enter.

   b. If you only want to scope the sampling to certain values for the attribute you just entered, select the attribute in the selection box, then enter a value in the **Value** box, for example, Seattle and press enter. You can continue to add multiple values for attributes. Just ensure that the correct attribute is selected when you're entering values.

   For example, if you include an **Attributes** value of City, and you want to limit your table to only include rows with a city value of New York and Dubai, you would enter City in the Attributes box, and New York and then Dubai in the **Values** box.

4. Click **OK**.
5. After completing the mapping definitions for the collections you want to sample, in the **Schema Editor** window, click **Sample**. For each column, you can modify the column SQL name, the SQL type, SQL length (if applicable), Scale (if applicable), Precision (if applicable) and Nullable.
   - You can set **Hide Column** to **true** if you want to exclude that column from query results. Columns marked Hide Column = true are not returned for selection and projection, although they are still part of the schema. For example, you can hide all the Azure Cosmos DB system required properties starting with "_".

- The **id** column is the only field that cannot be hidden as it is used as the primary key in the normalized schema.

6. Once you have finished defining the schema, click **File** | **Save**, navigate to the directory to save the schema, and then click **Save**.
7. Back in the **Azure Cosmos DB ODBC Driver DSN Setup** window, click ** Advanced Options**. Then, in the **Schema File** box, navigate to the saved schema file and click **OK**. Click **OK** again to save the DSN. This saves the schema you created to the DSN.

(Optional) Creating views

You can define and create views as part of the sampling process. These views are equivalent to SQL views. They are read-only and are scope the selections and projections of the Azure Cosmos DB SQL defined.

To create a view for your data, in the **Schema Editor** window, in the **View Definitions** column, click **Add** on the row of the collection to sample. Then in the **View Definitions** window, do the following:

1. Click **New**, enter a name for the view, for example, EmployeesfromSeattleView and then click **OK**.
2. In the **Edit view** window, enter an Azure Cosmos DB query. This must be an Azure Cosmos DB SQL query, for example `SELECT c.City, c.EmployeeName, c.Level, c.Age, c.Gender, c.Manager FROM c WHERE c.City = “Seattle”`, and then click **OK**.

You can create a many views as you like. Once you are done defining the views, you can then sample the data.

Step 5: View your data in BI tools such as Power BI Desktop

You can use your new DSN to connect DocumentADB with any ODBC-compliant tools - this step simply shows you how to connect to Power BI Desktop and create a Power BI visualization.

1. Open Power BI Desktop.
2. Click **Get Data**.
3. In the **Get Data** window, click **Other** | **ODBC** | **Connect**.
4. In the **From ODBC** window, select the data source name you created, and then click **OK**. You can leave the **Advanced Options** entries blank.
5. In the **Access a data source using an ODBC driver** window, select **Default or Custom** and then click **Connect**. You do not need to include the **Credential connection string properties**.
6. In the **Navigator** window, in the left pane, expand the database, the schema, and then select the table. The results pane includes the data using the schema you created.
7. To visualize the data in Power BI desktop, check the box in front of the table name, and then click **Load**.
8. In Power BI Desktop, on the far left, select the Data tab ▦ to confirm your data was imported.
9. You can now create visuals using Power BI by clicking on the Report tab 📊, clicking **New Visual**, and then customizing your tile. For more information about creating visualizations in Power BI Desktop, see Visualization types in Power BI.

Troubleshooting

If you receive the following error, ensure the **Host** and **Access Key** values you copied the Azure portal in Step 2 are correct and then retry. Use the copy buttons to the right of the **Host** and **Access Key** values in the Azure portal to copy the values error free.

```
[HY000]: [Microsoft][Azure Cosmos DB] (401) HTTP 401 Authentication Error: {"code":"Unauthorized","message":"The input authorization token can't serve the request. Please check that the expected payload is built as per the protocol, and check the key being used. Server used the following payload to sign: 'get\ndbs\n\nfri, 20 jan 2017 03:43:55 gmt\n\n'\r\nActivityId: 9acb3c0d-cb31-4b78-ac0a-413c8d33e373"}`
```

Next steps

To learn more about Azure Cosmos DB, see What is Azure Cosmos DB?.

DocumentDB Java SDK: Release notes and resources

5/30/2017 • 6 min to read • Edit Online

SDK Download	Maven
API documentation	Java API reference documentation
Contribute to SDK	GitHub
Get started	Get started with the Java SDK
Web app tutorial	Web application development with DocumentDB
Current supported runtime	JDK 7

Release Notes

1.11.0

- Added support for Request Unit per Minute (RU/m) feature.
- Added support for a new consistency level called ConsistentPrefix.
- Fixed a bug in reading collection in session mode.

1.10.0

- Enabled support for partitioned collection with as low as 2,500 RU/sec and scale in increments of 100 RU/sec.
- Fixed a bug in the native assembly which can cause NullRef exception in some queries.

1.9.6

- Fixed a bug in the query engine configuration that may cause exceptions for queries in Gateway mode.
- Fixed a few bugs in the session container that may cause an "Owner resource not found" exception for requests immediately after collection creation.

1.9.5

- Added support for aggregation queries (COUNT, MIN, MAX, SUM, and AVG). See Aggregation support.
- Added support for change feed.
- Added support for collection quota information through RequestOptions.setPopulateQuotaInfo.
- Added support for stored procedure script logging through RequestOptions.setScriptLoggingEnabled.
- Fixed a bug where query in DirectHttps mode may hang when encountering throttle failures.
- Fixed a bug in session consistency mode.
- Fixed a bug which may cause NullReferenceException in HttpContext when request rate is high.
- Improved performance of DirectHttps mode.

1.9.4

- Added simple client instance-based proxy support with ConnectionPolicy.setProxy() API.
- Added DocumentClient.close() API to properly shutdown DocumentClient instance.
- Improved query performance in direct connectivity mode by deriving the query plan from the native assembly

instead of the Gateway.

- Set FAIL_ON_UNKNOWN_PROPERTIES = false so users don't need to define JsonIgnoreProperties in their POJO.
- Refactored logging to use SLF4J.
- Fixed a few other bugs in consistency reader.

1.9.3

- Fixed a bug in the connection management to prevent connection leaks in direct connectivity mode.
- Fixed a bug in the TOP query where it may throw NullReferenece exception.
- Improved performance by reducing the number of network call for the internal caches.
- Added status code, ActivityID and Request URI in DocumentClientException for better troubleshooting.

1.9.2

- Fixed an issue in the connection management for stability.

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 Cosmos DB 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
1.11.0	May 10, 2017	---
1.10.0	March 11, 2017	---
1.9.6	February 21, 2017	---
1.9.5	January 31, 2017	---
1.9.4	November 24, 2016	---
1.9.3	October 30, 2016	---

VERSION	RELEASE DATE	RETIREMENT DATE
1.9.2	October 28, 2016	---
1.9.1	October 26, 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 Cosmos DB Team and request their assistance before the cutoff date.

See Also

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

DocumentDB .NET SDK: Download and release notes

6/12/2017 • 10 min to read • Edit Online

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 Azure Cosmos DB
Current supported framework	Microsoft .NET Framework 4.5

Release notes

1.14.1

- Fixed an issue that affected x64 machines that don't support SSE4 instruction and throw SEHException when running DocumentDB queries.

1.14.0

- Added support for Request Unit per Minute (RU/m) feature.
- Added support for a new consistency level called ConsistentPrefix.
- Added support for query metrics for individual partitions.
- Added support for limiting the size of the continuation token for queries.
- Added support for more detailed tracing for failed requests.
- Made some performance improvements in the SDK.

1.13.4

- Functionally same as 1.13.3. Made some internal changes.

1.13.3

- Functionally same as 1.13.2. Made some internal changes.

1.13.2

- Fixed an issue that ignored the PartitionKey value provided in FeedOptions for aggregate queries.
- Fixed an issue in transparent handling of partition management during mid-flight cross-partition Order By query execution.

1.13.1

- Fixed an issue which caused deadlocks in some of the async APIs when used inside ASP.NET context.

1.13.0

- Fixes to make SDK more resilient to automatic failover under certain conditions.

1.12.2

- Fix for an issue that occasionally causes a WebException: The remote name could not be resolved.
- Added the support for directly reading a typed document by adding new overloads to ReadDocumentAsync API.

1.12.1

- Added LINQ support for aggregation queries (COUNT, MIN, MAX, SUM, and AVG).
- Fix for a memory leak issue for the ConnectionPolicy object caused by the use of event handler.
- Fix for an issue wherein UpsertAttachmentAsync was not working when ETag was used.
- Fix for an issue wherein cross partition order-by query continuation was not working when sorting on string field.

1.12.0

- Added support for aggregation queries (COUNT, MIN, MAX, SUM, and AVG). See Aggregation support.
- Lowered minimum throughput on partitioned collections from 10,100 RU/s to 2500 RU/s.

1.11.4

- Fix for an issue wherein some of the cross-partition queries were failing in the 32-bit host process.
- Fix for an issue wherein the session container was not being updated with the token for failed requests in Gateway mode.
- Fix for an issue wherein a query with UDF calls in projection was failing in some cases.
- Client side performance fixes for increasing the read and write throughput of the requests.

1.11.3

- Fix for an issue wherein the session container was not being updated with the token for failed requests.
- Added support for the SDK to work in a 32-bit host process. Note that if you use cross partition queries, 64-bit host processing is recommended for improved performance.
- Improved performance for scenarios involving queries with a large number of partition key values in an IN expression.
- Populated various resource quota stats in the ResourceResponse for document collection read requests when PopulateQuotaInfo request option is set.

1.11.1

- Minor performance fix for the CreateDocumentCollectionIfNotExistsAsync API introduced in 1.11.0.
- Performance fix in the SDK for scenarios that involve high degree of concurrent requests.

1.11.0

- Support for new classes and methods to process the change feed of documents within a collection.
- Support for cross-partition query continuation and some perf improvements for cross-partition queries.
- Addition of CreateDatabaseIfNotExistsAsync and CreateDocumentCollectionIfNotExistsAsync methods.
- LINQ support for system functions: IsDefined, IsNull and IsPrimitive.
- Fix for automatic binplacing of Microsoft.Azure.Documents.ServiceInterop.dll and DocumentDB.Spatial.Sql.dll assemblies to application’s bin folder when using the Nuget package with projects that have project.json tooling.
- Support for emitting client side ETW traces which could be helpful in debugging scenarios.

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 DocumenClient 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 ExecuteStoredProcedureAsync 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 Cosmos DB
- **[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

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 Cosmos DB using a retired SDK will be rejected by the service.

VERSION	RELEASE DATE	RETIREMENT DATE
1.14.1	May 23, 2017	---

VERSION	RELEASE DATE	RETIREMENT DATE
1.14.0	May 10, 2017	---
1.13.4	May 09, 2017	---
1.13.3	May 06, 2017	---
1.13.2	April 19, 2017	---
1.13.1	March 29, 2017	---
1.13.0	March 24, 2017	---
1.12.2	March 20, 2017	---
1.12.1	March 14, 2017	---
1.12.0	February 15, 2017	---
1.11.4	February 06, 2017	---
1.11.3	January 26, 2017	---
1.11.1	December 21, 2016	---
1.11.0	December 08, 2016	---
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	---
1.8.0	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	---

VERSION	RELEASE DATE	RETIREMENT DATE
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	---

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 Cosmos DB Team and request their assistance before the cutoff date.

See also

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

DocumentDB .NET Core SDK: Release notes and resources

6/12/2017 • 2 min to read • Edit Online

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 and .NET Standard 1.5

Release Notes

The DocumentDB .NET Core SDK has feature parity with the latest version of the DocumentDB .NET SDK.

> NOTE
>
> The DocumentDB .NET Core SDK is not yet compatible with Universal Windows Platform (UWP) apps. If you are interested in the .NET Core SDK that does support UWP apps, send email to askcosmosdb@microsoft.com.

1.3.2

- Supporting .NET Standard 1.5 as one of the target frameworks.

1.3.1

- Fixed an issue that affected x64 machines that don't support SSE4 instruction and throw SEHException when running DocumentDB queries.

1.3.0

- Added support for Request Unit per Minute (RU/m) feature.
- Added support for a new consistency level called ConsistentPrefix.
- Added support for query metrics for individual partitions.
- Added support for limiting the size of the continuation token for queries.
- Added support for more detailed tracing for failed requests.
- Made some performance improvements in the SDK.

1.2.2

- Fixed an issue that ignored the PartitionKey value provided in FeedOptions for aggregate queries.
- Fixed an issue in transparent handling of partition management during mid-flight cross-partition Order By query execution.

1.2.1

- Fixed an issue which caused deadlocks in some of the async APIs when used inside ASP.NET context.

1.2.0

- Fixes to make SDK more resilient to automatic failover under certain conditions.

1.1.2

- Fix for an issue that occasionally causes a WebException: The remote name could not be resolved.
- Added the support for directly reading a typed document by adding new overloads to ReadDocumentAsync API.

1.1.1

- Added LINQ support for aggregation queries (COUNT, MIN, MAX, SUM, and AVG).
- Fix for a memory leak issue for the ConnectionPolicy object caused by the use of event handler.
- Fix for an issue wherein UpsertAttachmentAsync was not working when ETag was used.
- Fix for an issue wherein cross partition order-by query continuation was not working when sorting on string field.

1.1.0

- Added support for aggregation queries (COUNT, MIN, MAX, SUM, and AVG). See Aggregation support.
- Lowered minimum throughput on partitioned collections from 10,100 RU/s to 2500 RU/s.

1.0.0

The DocumentDB .NET Core SDK enables you to build fast, cross-platform ASP.NET Core and .NET Core apps to run on Windows, Mac, and Linux. The latest release of the DocumentDB .NET Core SDK is fully Xamarin compatible and be used to build applications that target iOS, Android, and Mono (Linux).

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.

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, or file an issue in the github repository.

Release & Retirement Dates

VERSION	RELEASE DATE	RETIREMENT DATE
1.3.2	June 12, 2017	---
1.3.1	May 23, 2017	---
1.3.0	May 10, 2017	---
1.2.2	April 19, 2017	---
1.2.1	March 29, 2017	---

VERSION	RELEASE DATE	RETIREMENT DATE
1.2.0	March 25, 2017	---
1.1.2	March 20, 2017	---
1.1.1	March 14, 2017	---
1.1.0	February 16, 2017	---
1.0.0	December 21, 2016	---
0.1.0-preview	November 15, 2016	December 31, 2016

See Also

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

DocumentDB Node.js SDK: Release notes and resources

5/30/2017 • 6 min to read • Edit Online

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

1.12.0

- Added support for Request Unit per Minute (RU/m) feature.
- Added support for a new consistency level called ConsistentPrefix.
- Added support for UriFactory.
- Fixed a unicode support bug. (GitHub issue #171)

1.11.0

- Added the support for aggregation queries (COUNT, MIN, MAX, SUM, and AVG).
- Added the option for controlling degree of parallelism for cross partition queries.
- Added the option for disabling SSL verification when running against DocumentDB Emulator.
- Lowered minimum throughput on partitioned collections from 10,100 RU/s to 2500 RU/s.
- Fixed the continuation token bug for single partition collection (github #107).
- Fixed the executeStoredProcedure bug in handling 0 as single param (github #155).

1.10.2

- Fixed user-agent header to include the SDK version.
- Minor code cleanup.

1.10.1

- Disabling SSL verification when using the SDK to target the emulator(hostname=localhost).
- Added support for enabling script logging during stored procedure execution.

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 overriden in the RetryOptions property on ConnectionPolicy object.
- Cosmos DB 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 cummulative 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 hashParitionResolver 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 Cosmos DB using a retired SDK will be rejected by the service.

VERSION	RELEASE DATE	RETIREMENT DATE
1.12.0	May 10, 2017	---
1.11.0	March 16, 2017	---
1.10.2	January 27, 2017	---
1.10.1	December 22, 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	---
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	---

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 Cosmos DB Team and request their assistance before the cutoff date.

See also

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

DocumentDB Python SDK: Release notes and resources

5/30/2017 • 5 min to read • Edit Online

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

2.2.0

- Added support for Request Unit per Minute (RU/m) feature.
- Added support for a new consistency level called ConsistentPrefix.

2.1.0

- Added support for aggregation queries (COUNT, MIN, MAX, SUM, and AVG).
- Added an option for disabling SSL verification when running against DocumentDB Emulator.
- Removed the restriction of dependent requests module to be exactly 2.10.0.
- Lowered minimum throughput on partitioned collections from 10,100 RU/s to 2500 RU/s.
- Added support for enabling script logging during stored procedure execution.
- REST API version bumped to '2017-01-19' with this release.

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 overriden in the RetryOptions property on ConnectionPolicy object.

- Cosmos DB 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 cummulative 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 Cosmos DB 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
2.2.0	May 10, 2017	---
2.1.0	May 01, 2017	---
2.0.1	October 30, 2016	---
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 Cosmos DB Team and request their assistance before the cutoff date.

See also

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

Azure Cosmos DB Table .NET API: Download and release notes

5/30/2017 • 1 min to read • Edit Online

SDK download	NuGet
API documentation	.NET API reference documentation
Quickstart	Azure Cosmos DB: Build an app with .NET and the Table API
Tutorial	Azure CosmosDB: Create a container with the Graph API
Current supported framework	Microsoft .NET Framework 4.5

Release notes

- Initial preview release.

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 Azure Cosmos DB using a retired SDK will be rejected by the service.

See also

To learn more about the Azure Cosmos DB Table API, see Introduction to Azure Cosmos DB: Table API.

Azure Cosmos DB Graph .NET API: Download and release notes

5/30/2017 • 1 min to read • Edit Online

SDK download	NuGet
API documentation	.NET API reference documentation
Quickstart	Azure Cosmos DB: Create a graph app using .NET and the Graph API
Tutorial	Azure CosmosDB: Create a container with the Graph API
Current supported framework	Microsoft .NET Framework 4.5

Release notes

- Initial preview release.

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 Azure Cosmos DB using a retired SDK will be rejected by the service.

See also

To learn more about the Azure Cosmos DB Graph API, see Introduction to Azure Cosmos DB: Graph API.

Azure Cosmos DB FAQ

6/13/2017 • 29 min to read • Edit Online

Azure Cosmos DB fundamentals

What is Azure Cosmos DB?

Azure Cosmos DB is a globally replicated, multi-model database service that that offers rich querying over schema-free data, helps deliver configurable and reliable performance, and enables rapid development. It's all achieved through a managed platform that's backed by the power and reach of Microsoft Azure.

Azure Cosmos DB 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. It delivers schema flexibility and rich indexing, and it includes multi-document transactional support with integrated JavaScript.

For more database questions, answers, and instructions for deploying and using this service, see the Azure Cosmos DB documentation page.

What happened to DocumentDB?

The DocumentDB API is one of the supported APIs and data models for Azure Cosmos DB. In addition, Azure Cosmos DB supports you with Graph API (Preview), Table API (Preview) and MongoDB API. For more information, see Questions from DocumentDB customers.

How do I get to my DocumentDB account in the Azure portal?

In the Azure portal, click the Azure Cosmos DB icon in the left pane. If you had a DocumentDB account before, you now have an Azure Cosmos DB account, with no change to your billing.

What are the typical use cases for Azure Cosmos DB?

Azure Cosmos DB 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. Azure Cosmos DB 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 Azure Cosmos DB.

How does Azure Cosmos DB offer predictable performance?

A request unit (RU) is the measure of throughput in Azure Cosmos DB. A 1-RU throughput corresponds to the throughput of the GET of a 1-KB document. Every operation in Azure Cosmos DB, including reads, writes, SQL queries, and stored procedure executions, has a deterministic RU value that's based on the throughput required to complete the operation. Instead of thinking about CPU, IO, and memory and how they each affect your application throughput, you can think in terms of a single RU measure.

You can reserve each Azure Cosmos DB container 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 a container to handle the total of request units across all requests. You can also scale up or scale down your container's throughput as the needs of your application evolve. For more information about request units and for help determining your container needs, see Estimating throughput needs and try the throughput calculator. The term *container* here refers to refers to a DocumentDB API collection, Graph API graph, MongoDB API collection, and Table API table.

Is Azure Cosmos DB HIPAA compliant?

Yes, Azure Cosmos DB 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 Azure Cosmos DB?

There is no limit to the total amount of data that a container can store in Azure Cosmos DB.

What are the throughput limits of Azure Cosmos DB?

There is no limit to the total amount of throughput that a container can support in Azure Cosmos DB. The key idea is to distribute your workload roughly evenly among a sufficiently large number of partition keys.

How much does Azure Cosmos DB cost?

For details, refer to the Azure Cosmos DB pricing details page. Azure Cosmos DB usage charges are determined by the number of provisioned containers, the number of hours the containers were online, and the provisioned throughput for each container. The term *containers* here refers to the DocumentDB API collection, Graph API graph, MongoDB API collection, and Table API tables.

Is 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 Cosmos DB 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 Azure Cosmos DB Emulator, you can switch to using an Azure Cosmos DB account in the cloud.

How can I get additional help with Azure Cosmos DB?

If you need any help, reach out to us on Stack Overflow or the MSDN forum, or schedule a one-on-one chat with the Azure Cosmos DB engineering team by sending mail to askcosmosdb@microsoft.com.

Set up Azure Cosmos DB

How do I sign up for Azure Cosmos DB?

Azure Cosmos DB is available in the Azure portal. First, sign up for an Azure subscription. After you've signed up, you can add a DocumentDB API, Graph API (Preview), Table API (Preview), or MongoDB API account to your Azure subscription.

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 all resources in the database account. Use caution when you distribute master keys. The primary master key and secondary master key are available on the **Keys** blade of the Azure portal. For more information about keys, see View, copy, and regenerate access keys.

What are the regions that PreferredLocations can be set to?

The PreferredLocations value can be set to any of the Azure regions in which Cosmos DB is available. For a list of available regions, see Azure regions.

Is there anything I should be aware of when distributing data across the world via the Azure datacenters?

Azure Cosmos DB is present across all Azure regions, as specified on the Azure regions page. Because it is the core service, every new datacenter has an Azure Cosmos DB presence.

When you set a region, remember that Azure Cosmos DB respects sovereign and government clouds. That is, if you create an account in a sovereign region, you cannot replicate out of that sovereign region. Similarly, you cannot enable replication into other sovereign locations from an outside account.

Develop against the DocumentDB API

How do I start developing against the DocumentDB API?

Microsoft DocumentDB API is available in the Azure portal. First you must sign up for an Azure subscription. Once you sign up for an Azure subscription, you can add DocumentDB API container to your Azure subscription. For instructions on adding an Azure Cosmos DB account, see Create an Azure Cosmos DB database account. If you had a DocumentDB account in the past, you now have an Azure Cosmos DB account.

SDKs are available for .NET, Python, Node.js, JavaScript, and Java. Developers can also use the RESTful HTTP APIs to interact with Azure Cosmos DB resources from various platforms and languages.

Can I access some ready-made samples to get a head start?

Samples for the DocumentDB API .NET, Java, Node.js, and Python SDKs are available on GitHub.

Does the DocumentDB API database support schema-free data?

Yes, the DocumentDB API allows applications to store arbitrary JSON documents without schema definitions or hints. Data is immediately available for query through the Azure Cosmos DB SQL query interface.

Does the DocumentDB API support ACID transactions?

Yes, the DocumentDB API 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 is a collection?

A collection is a group of documents and their associated JavaScript application logic. A collection is a billable entity, where the cost is determined by the throughput and used storage. Collections can span one or more partitions or servers and can scale to handle practically unlimited volumes of storage or throughput.

Collections are also the billing entities for Azure Cosmos DB. Each collection is billed hourly, based on the provisioned throughput and used storage space. For more information, see DocumentDB API pricing.

How do I create a database?

You can create databases by using the Azure portal, as described in Add a collection, one of the Azure Cosmos DB SDKs, or the REST APIs.

How do I set up users and permissions?

You can create users and permissions by using one of the DocumentDB API SDKs or the REST APIs.

Does the DocumentDB API support SQL?

The SQL query language is an enhanced subset of the query functionality that's supported by SQL. The Azure Cosmos DB 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 labeled nodes, which are used by both the Azure Cosmos DB automatic indexing techniques and the SQL query dialect of Azure Cosmos DB. For information about using SQL grammar, see the QueryDocumentDB article.

Does the DocumentDB API support SQL aggregation functions?

The DocumentDB API supports low-latency aggregation at any scale via aggregate functions `COUNT`, `MIN`, `MAX`, `AVG`, and `SUM` via the SQL grammar. For more information, see Aggregate functions.

How does the DocumentDB API provide concurrency?

The DocumentDB API supports optimistic concurrency control (OCC) through HTTP entity tags, or ETags. Every DocumentDB API 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 whether 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 is updated. If the ETag is no longer current, the server rejects the operation with an "HTTP 412 Precondition failure" response code. The client then re-fetches the resource to acquire the current ETag value for the resource. In addition, ETags can be used with the If-None-Match header to determine whether 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 the DocumentDB API?

The DocumentDB API supports language-integrated transactions via JavaScript-stored procedures and triggers. All database operations inside scripts are executed under snapshot isolation. If it is a single-partition collection, the execution is scoped to the collection. If the collection is partitioned, the execution is scoped to documents with the same partition-key value within the collection. 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. For more information, see DocumentDB API server-side programming.

How can I bulk-insert documents into the DocumentDB API?

You can bulk-insert documents into Azure Cosmos DB in either of two ways:

- The data migration tool, as described in Import data to DocumentDB API.
- Stored procedures, as described in DocumentDB API server-side programming.

Does the DocumentDB API support resource link caching?

Yes, because Azure Cosmos DB is a RESTful service, resource links are immutable and can be cached. DocumentDB API clients can specify an "If-None-Match" header for reads against any resource-like document or collection and then update their local copies after the server version has changed.

Is a local instance of DocumentDB API available?

Yes. The Azure Cosmos DB Emulator provides a high-fidelity emulation of the DocumentDB API service. It supports functionality that's identical to Azure Cosmos DB, including support for creating and querying JSON documents, provisioning and scaling collections, and executing stored procedures and triggers. You can develop and test applications by using the Azure Cosmos DB Emulator, and deploy them to Azure at a global scale by making a single configuration change to the connection endpoint for Azure Cosmos DB.

Develop against the API for MongoDB

What is the Azure Cosmos DB API for MongoDB?

The Azure Cosmos DB API for MongoDB is a compatibility layer that allows applications to easily and transparently communicate with the native Azure Cosmos DB database engine by using existing, community-supported Apache MongoDB APIs and drivers. Developers can now use existing MongoDB tool chains and skills to build applications that take advantage of Azure Cosmos DB. Developers benefit from the unique capabilities of Azure Cosmos DB, which include auto-indexing, backup maintenance, financially backed service level agreements (SLAs), and so on.

How do I connect to my API for MongoDB database?

The quickest way to connect to the Azure Cosmos DB API for MongoDB is to head over to the Azure portal. Go to your account and then, on the left navigation menu, click **Quick Start**. Quick Start is the best way to get code snippets to connect to your database.

Azure Cosmos DB enforces strict security requirements and standards. Azure Cosmos DB accounts require authentication and secure communication via SSL, so be sure to use TLSv1.2.

For more information, see Connect to your API for MongoDB database.

Are there additional error codes for an API for MongoDB database?

In addition to the common MongoDB error codes, the MongoDB API has its own specific error codes:

ERROR	CODE	DESCRIPTION	SOLUTION
TooManyRequests	16500	The total number of request units consumed has exceeded the provisioned request-unit rate for the collection and has been throttled.	Consider scaling the throughput of the collection from the Azure portal or retrying again.
ExceededMemoryLimit	16501	As a multi-tenant service, the operation has exceeded the client's memory allotment.	Reduce the scope of the operation through more restrictive query criteria or contact support from the Azure portal. Example: *db.getCollection('users').aggregate([* *{$match: {name: "Andy"}},* *{$sort: {age: -1}}* *]))*

Develop with the Table API (Preview)

Terms

The Azure Cosmos DB: Table API (Preview) refers to the premium offering by Azure Cosmos DB for table support announced at Build 2017.

The standard table SDK is the existing Azure Storage table SDK.

How can I use the new Table API (Preview) offering?

The Azure Cosmos DB Table API is available in the Azure portal. First you must sign up for an Azure subscription. After you've signed up, you can add an Azure Cosmos DB Table API account to your Azure subscription, and then add tables to your account.

During the preview period, when SDKs are available for .NET, you can get started by completing the Table API quick-start article.

Do I need a new SDK to use the Table API (Preview)?

Yes, the Windows Azure Storage premium table (Preview) SDK is available on NuGet. Additional information is available on the Azure Cosmos DB Table .NET API: Download and release notes page.

How do I provide feedback about the SDK or bugs?

You can share your feedback in any of the following ways:

- Uservoice
- MSDN forum
- Stackoverflow

What is the connection string that I need to use to connect to the Table API (Preview)?

The connection string is `DefaultEndpointsProtocol=https;AccountName=<AccountNamefromCosmos DB;AccountKey=<FromKeysPaneofCosmosDB>;TableEndpoint=https://<AccountNameFromDocumentDB>.documents.azure.com`. You can get the connection string from the Keys page in the Azure portal.

How do I override the config settings for the request options in the new Table API (Preview)?

For information about config settings, see Azure Cosmos DB capabilities. You can change the settings by adding them to app.config in the appSettings section in the client application.

```
<appSettings>
   <add key="TableConsistencyLevel" value="Eventual|Strong|Session|BoundedStaleness|ConsistentPrefix"/>
   <add key="TableThroughput" value="<PositiveIntegerValue"/>
   <add key="TableIndexingPolicy" value="<jsonindexdefn>"/>
   <add key="TableUseGatewayMode" value="True|False"/>
   <add key="TablePreferredLocations" value="Location1|Location2|Location3|Location4>"/>....
</appSettings>
```

Are there any changes for customers who are using the existing standard table SDK?

None. There are no changes for existing or new customers who are using the existing standard table SDK.

How do I view table data that is stored in Azure Cosmos DB for use with the Table API (review)?

You can use the Azure portal to browse the data. You can also use the Table API (Preview) code or the tools mentioned in the next answer.

Which tools work with the Table API (Preview)?

You can use the older version of Azure Explorer (0.8.9).

Tools with the flexibility to take a connection string in the format specified previously can support the new Table API (Preview). A list of table tools is provided on the Azure Storage Client Tools page.

Do PowerShell or Azure CLI work with the new Table API (Preview)?

We plan to add support for PowerShell and Azure CLI for Table API (Preview).

Is the concurrency on operations controlled?

Yes, optimistic concurrency is provided via the use of the ETag mechanism.

Is the OData query model supported for entities?

Yes, the Table API (Preview) supports OData query and LINQ query.

Can I connect to the standard Azure table and the new premium Table API (Preview) side by side in the same application?

Yes, you can connect by creating two separate instances of the CloudTableClient, each pointing to its own URI via the connection string.

How do I migrate an existing Azure Table storage application to this new offering?

To take advantage of the new Table API offering on your existing Table storage data, contact askcosmosdb@microsoft.com.

What is the roadmap for this service, and when will you offer other standard Table API functionality?

We plan to add support for SAS tokens, ServiceContext, Stats, Encryption, Analytics, and other features as we proceed toward GA. You can give us feedback on Uservoice.

How is expansion of the storage size done for this service if, for example, I start with *n* GB of data and my data will grow to 1 TB over time?

Azure Cosmos DB is designed to provide unlimited storage via the use of horizontal scaling. The service can monitor and effectively increase your storage.

How do I monitor the Table API (Preview) offering?

You can use the Table API (Preview) **Metrics** pane to monitor requests and storage usage.

How do I calculate the throughput I require?

You can use the capacity estimator to calculate the TableThroughput that's required for the operations. For more information, see Estimate Request Units and Data Storage. In general, you can represent your entity as JSON and provide the numbers for your operations.

Can I use the new Table API (Preview) SDK locally with the emulator?

Yes, you can use the Table API (Preview) with the local emulator when you use the new SDK. To download new emulator, go to Use the Azure Cosmos DB Emulator for local development and testing. The StorageConnectionString value in app.config needs to be

```
DefaultEndpointsProtocol=https;AccountName=localhost;AccountKey=C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==;TableEndpoint=https://localhost:80
```

.

Can my existing application work with the Table API (Preview)?

The surface area of the new Table API (Preview) is compatible with the existing Azure standard table SDK across the create, delete, update, and query operations in the .NET API. Ensure that you have a row key, because the Table API (Preview) requires both a partition key and a row key. We also plan to add more SDK support as we proceed toward GA of this service offering.

Do I need to migrate my existing Azure table-based applications to the new SDK if I do not want to use the Table API (Preview) features?

No, you can create and use existing standard table assets without interruption of any kind. However, if you do not use the new Table API (Preview), you cannot benefit from the automatic index, the additional consistency option, or global distribution.

How do I add replication of the data in the premium Table API (Preview) across multiple regions of Azure?

You can use the Azure Cosmos DB portal's global replication settings to add regions that are suitable for your application. To develop a globally distributed application, you should also add your application with the PreferredLocation information set to the local region for providing low read latency.

How do I change the primary write region for the account in the premium Table API (Preview)?

You can use the Azure Cosmos DB global replication portal pane to add a region and then fail over to the required region. For instructions, see Developing with multi-region Azure Cosmos DB accounts.

How do I configure my preferred read regions for low latency when I distribute my data?

To help read from the local location, use the PreferredLocation key in the app.config file. For existing applications, the Table API (Preview) throws an error if LocationMode is set. Remove that code, because the premium Table API (Preview) picks up this information from the app.config file. For more information, see Azure Cosmos DB capabilities.

How should I think about consistency levels in the Table API (Preview)?

Azure Cosmos DB provides well-reasoned trade-offs between consistency, availability, and latency. Azure Cosmos DB offers five consistency levels to Table API

(Preview) developers, so you can choose the right consistency model at the table level and make individual requests while querying the data. When a client connects, it can specify a consistency level. You can change the level via the app.config setting for the value of the TableConsistencyLevel key.

The Table API (Preview) provides low-latency reads with "Read your own writes," with Session consistency as the default. For more information, see Consistency levels.

By default, Azure Table storage provides Strong consistency within a region and Eventual consistency in the secondary locations.

Does Azure Cosmos DB offer more consistency levels than standard tables?

Yes, for information about how to benefit from the distributed nature of Azure Cosmos DB, see Consistency levels. Because guarantees are provided for the consistency levels, you can use them with confidence. For more information, see Azure Cosmos DB capabilities.

When global distribution is enabled, how long does it take to replicate the data?

We commit the data durably in the local region and push the data to other regions immediately in a matter of milliseconds. This replication is dependent only on the round-trip time (RTT) of the datacenter. To learn more about the global-distribution capability of Azure Cosmos DB, see Azure Cosmos DB: A globally distributed database service on Azure.

Can the read request consistency level be changed?

With Azure Cosmos DB, you can set the consistency level at the container level (on the table). By using the SDK, you can change the level by providing the value for TableConsistencyLevel key in the app.config file. The possible values are: Strong, Bounded Staleness, Session, Consistent Prefix, and Eventual. For more information, see Tunable data consistency levels in Azure Cosmos DB. The key idea is that you cannot set the request consistency level at more than the setting for the table. For example, you cannot set the consistency level for the table at Eventual and the request consistency level at Strong.

How does the premium Table API (Preview) account handle failover if a region goes down?

The premium Table API (Preview) borrows from the globally distributed platform of Azure Cosmos DB. To ensure that your application can tolerate datacenter downtime, enable at least one more region for the account in the Azure Cosmos DB portal Developing with multi-region Azure Cosmos DB accounts. You can set the priority of the region by using the portal Developing with multi-region Azure Cosmos DB accounts.

You can add as many regions as you want for the account and control where it can fail over to by providing a failover priority. Of course, to use the database, you need to provide an application there too. When you do so, your customers will not experience downtime. The client SDK is auto homing. That is, it can detect the region that's down and automatically fail over to the new region.

Is the premium Table API (Preview) enabled for backups?

Yes, the premium Table API (Preview) borrows from the platform of Azure Cosmos DB for backups. Backups are made automatically. For more information, see Online backup and restore with Azure Cosmos DB.

Does the Table API (Preview) index all attributes of an entity by default?

Yes, all attributes of an entity are indexed by default. For more information, see Azure Cosmos DB: Indexing policies.

Does this mean I do not have to create multiple indexes to satisfy the queries?

Yes, Azure Cosmos DB provides automatic indexing of all attributes without any schema definition. This automation frees developers to focus on the application rather than on index creation and management. For more information, see Azure Cosmos DB: Indexing policies.

Can I change the indexing policy?

Yes, you can change the indexing policy by providing the index definition. For more information, see Azure Cosmos DB capabilities. You need to properly encode and escape the settings.

In string json format in the app.config file:

```
{
 "indexingMode": "consistent",
 "automatic": true,
 "includedPaths": [
   {
    "path": "/somepath",
    "indexes": [
      {
       "kind": "Range",
       "dataType": "Number",
       "precision": -1
      },
      {
       "kind": "Range",
       "dataType": "String",
       "precision": -1
      }
    ]
   }
 ],
 "excludedPaths":
[
 {
    "path": "/anotherpath"
 }
]
}
```

Azure Cosmos DB as a platform seems to have lot of capabilities, such as sorting, aggregates, hierarchy, and other functionality. Will you be adding these capabilities to the Table API?

In preview, the Table API provides the same query functionality as Azure Table storage. Azure Cosmos DB also supports sorting, aggregates, geospatial query, hierarchy, and a wide range of built-in functions. We will provide additional functionality in the Table API in a future service update. For more information, see Azure Cosmos DB query.

When should I change TableThroughput for the Table API (Preview)?

You should change TableThroughput when either of the following conditions applies:

- You're performing an extract, transform, and load (ETL) of data, or you want to upload a lot of data in short amount of time.

- You need more throughput from the container at the back end. For example, you see that the used throughput is more than the provisioned throughput, and you are getting throttled. For more information, see Set throughput.

Can I scale up or scale down the throughput of my Table API (Preview) table?

Yes, you can use the Azure Cosmos DB portal's scale pane to scale the throughput. For more information, see Set throughput.

Can the premium Table API (Preview) take advantage of the RU-per-minute offering?

Yes, the premium Table API (Preview) borrows from the capabilities of Azure Cosmos DB to provide SLAs for performance, latency, availability, and consistency. This capability ensures that the table can use the RU-per-minute offering. For more information, see Request Units in Azure Cosmos DB. With this capability, customers can avoid provisioning for the peak and smooth out the spikes in the workload.

Is a default TableThroughput set for newly provisioned tables?

Yes, if you do not override the TableThroughput via app.config and do not use a pre-created container in Azure Cosmos DB, the service creates a table with throughput of 400.

Is there any change of pricing for existing customers of the standard Table API?

None. There is no change in price for existing standard Table API customers.

How is the price calculated for the Table API (Preview)?

The price depends on the allocated TableThroughput.

How do I handle any throttling on the tables in Table API (Preview) offering?

If the request rate exceeds the capacity of the provisioned throughput for the underlying container, you will get an error, and the SDK will retry the call by applying the retry policy.

Why do I need to choose a throughput apart from PartitionKey and RowKey to take advantage of the premium Table API (Preview) offering of Azure Cosmos DB?

Azure Cosmos DB sets a default throughput for your container if you do not provide one in the app.config file.

Azure Cosmos DB provides guarantees for performance and latency, with upper bounds on operation. This guarantee is possible when the engine can enforce governance on the tenant's operations. Setting TableThroughput ensures that you get the guaranteed throughput and latency, because the platform reserves this capacity and guarantees operational success.

By using the throughput specification, you can elastically change it to benefit from the seasonality of your application, meet the throughput needs, and save costs.

Azure Storage SDK has been very inexpensive for me, because I pay only to store the data, and I rarely query. The new Azure Cosmos DB offering seems to be charging me even though I have not performed a single transaction or stored anything. Can you please explain?

Azure Cosmos DB is designed to be a globally distributed, SLA-based system with guarantees for availability, latency, and throughput. When you reserve throughput in Azure Cosmos DB, it is guaranteed, unlike the throughput of other systems. Azure Cosmos DB provides additional capabilities that customers have requested, such as secondary indexes and global distribution. During the preview period, we provide a throughput-optimized model and, eventually, we plan to provide a storage-optimized model to meet our customers' needs.

I never get a "quota full" notification (indicating that a partition is full) when I ingest data into Table storage. With the Table API (Preview), I do get this message. Is this offering limiting me and forcing me to change my existing application?

Azure Cosmos DB is an SLA-based system that provides unlimited scale, with guarantees for latency, throughput, availability, and consistency. To ensure guaranteed premium performance, make sure that your data size and index are manageable and scalable. The 10-GB limit on the number of entities or items per partition key is to ensure that we provide great lookup and query performance. To ensure that your application scales well even for Azure Storage, we recommend that you *not* create a hot partition by storing all information in one partition and querying it.

So PartitionKey and RowKey are still required with the new Table API (Preview)?

Yes. Because the surface area of the Table API (Preview) is similar to that of the Table storage SDK, the partition key provides an efficient way to distribute the data. The row key is unique within that partition. The row key needs to be present and can't be null as in the standard SDK. The length of RowKey is 255 bytes and the length of PartitionKey is 100 bytes (soon to be increased to 1 KB).

What are the error messages for the Table API (Preview)?

Because this preview is compatible with the standard table, most of the errors will map to the errors from the standard table.

Why do I get throttled when I try to create lot of tables one after another in the Table API (Preview)?

Azure Cosmos DB is an SLA-based system that provides latency, throughput, availability and consistency guarantees. Because it is a provisioned system, it reserves resources to guarantee these requirements. The rapid rate of creation of tables is detected and throttled. We recommend that you look at the rate of creation of tables and lower it to less than 5 per minute. Remember that the Table API (Preview) is a provisioned system. The moment you provision it, you will begin to pay for it.

Develop against the Graph API (Preview)

How can I apply the functionality of Graph API (Preview) to Azure Cosmos DB?

You can use an extension library to apply the functionality of Graph API (Preview). This library is called Microsoft Azure Graphs, and it is available on NuGet.

It looks like you support the Gremlin graph traversal language. Do you plan to add more forms of query?

Yes, we plan to add other mechanisms for query in the future.

How can I use the new Graph API (Preview) offering?

To get started, complete the Graph API quick-start article.

Questions from DocumentDB customers

Why are you moving to Azure Cosmos DB?

Azure Cosmos DB is the next big leap in globally distributed, at-scale cloud databases. As a DocumentDB customer, you now have access to the breakthrough system and capabilities offered by Azure Cosmos DB.

Azure Cosmos DB started as "Project Florence" in 2010 to address the pain points faced by developers in building large-scale applications inside Microsoft. The challenges of building globally distributed apps are not unique to Microsoft, so we made the first generation of this technology available in 2015 to Azure developers in the form of Azure DocumentDB.

Since that time, we've added new features and introduced significant new capabilities. Azure Cosmos DB is the result. As a part of this release, DocumentDB customers, with their data, automatically and seamlessly become Azure Cosmos DB customers. These capabilities are in the areas of the core database engine, as well as global distribution, elastic scalability, and industry-leading, comprehensive SLAs. Specifically, we have evolved the Azure Cosmos DB database engine to efficiently map all popular data models, type systems, and APIs to the underlying data model of Azure Cosmos DB.

The current developer-facing manifestation of this work is the new support for Gremlin and Table storage APIs. And this is just the beginning. We plan to add other popular APIs and newer data models over time, with more advances in performance and storage at global scale.

It is important to point out that the DocumentDB SQL dialect has always been just one of the many APIs that the underlying Azure Cosmos DB can support. For developers who use a fully managed service such as Azure Cosmos DB, the only interface to the service is the APIs that are exposed by the service. Nothing really changes for existing DocumentDB customers. In Azure Cosmos DB, you get exactly the same SQL API that DocumentDB offers. And now (and in the future), you can access other previously inaccessible capabilities

Another manifestation of our continued work is the extended foundation for global and elastic scalability of throughput and storage. One of the very first manifestations of scalability is the RU/m, but we plan to announce additional capabilities that can help reduce costs for our customers for various workloads. We have made several foundational enhancements to the global distribution subsystem. One of the many such developer-facing features is the Consistent Prefix consistency model, which makes a total five well-defined consistency models. We will release many more interesting capabilities as they mature.

What do I need to do to ensure that my DocumentDB resources continue to run on Azure Cosmos DB?

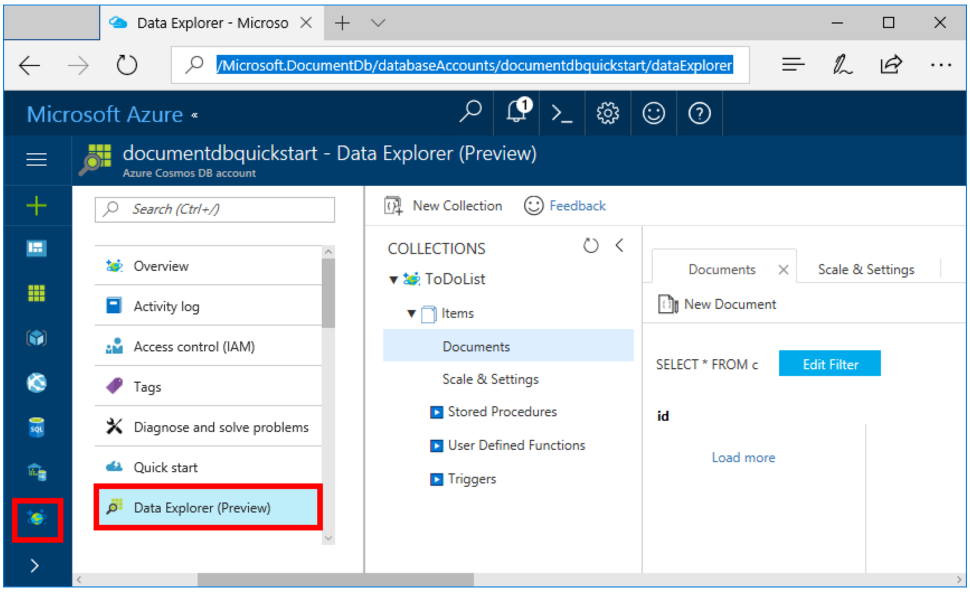
You need to make no changes at all. Your DocumentDB resources are now Azure Cosmos DB resources, and there was no interruption in the service when this move occurred.

What changes do I need to make for my app to work with Azure Cosmos DB?

There are no changes to make. Classes, namespaces, and NuGet package names have not changed. As always, we recommend that you keep your SDKs up to date to take advantage of the latest features and improvements.

What's changed in the Azure portal?

DocumentDB no longer appears in the portal as an Azure service. In its place is a new Azure Cosmos DB icon, as shown in the following image. All your collections are available, as they were before, and you can still scale throughput, change consistency levels, and monitor SLAs. The capabilities of Data Explorer (Preview) have been enhanced. You can now view and edit documents, create and run queries, and work with stored procedures, triggers, and UDF from one page, as shown in the following image:



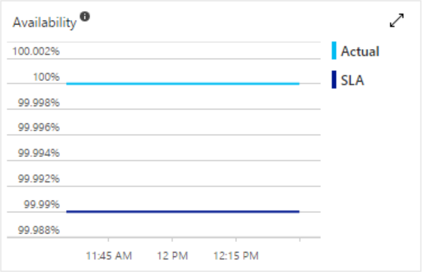
Are there changes to pricing?

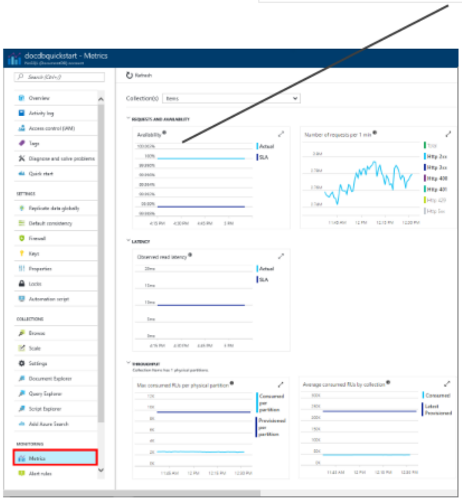
No, the cost of running your app on Azure Cosmos DB is the same as it was before. However, you might benefit from the new "Request Unit per minute" feature. For more information, see the Scale throughput per minute article.

Are there changes to the SLAs?

No, the SLAs for availability, consistency, latency, and throughput are unchanged and are still displayed in the portal. For more information, see SLA for Azure Cosmos DB.

Service Level Agreement (SLA) metrics are monitored and easy to revew in the Azure portal


Availability
11:45 AM
12 PM
12:15 PM
Actual
SLA


docdbquickstart - Metrics
Search (Ctrl+/)
Overview
Activity log
Access control (IAM)
Tags
Diagnose and solve problems
Quick start
SETTINGS
Replicate data globally
Default consistency
Firewall
Keys
Properties
Locks
Automation script
COLLECTIONS
Browse
Scale
Settings
Document Explorer
Query Explorer
Script Explorer
Add Azure Search
MONITORING
Metrics
Alert rules
Refresh
Collection(s)
Items
REQUESTS AND AVAILABILITY
Availability
Actual
SLA
Number of requests per 1 min
Http 2xx
Http 3xx
Http 400
Http 401
LATENCY
Observed read latency
Actual
SLA
THROUGHPUT
Max consumed RUs per physical partition
Consumed per partition
Provisioned per partition
Average consumed RUs by collection
Consumed
Latest Provisioned
11:45 AM
12 PM
12:15 PM