# DocumentDB hierarchical resource model and concepts

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

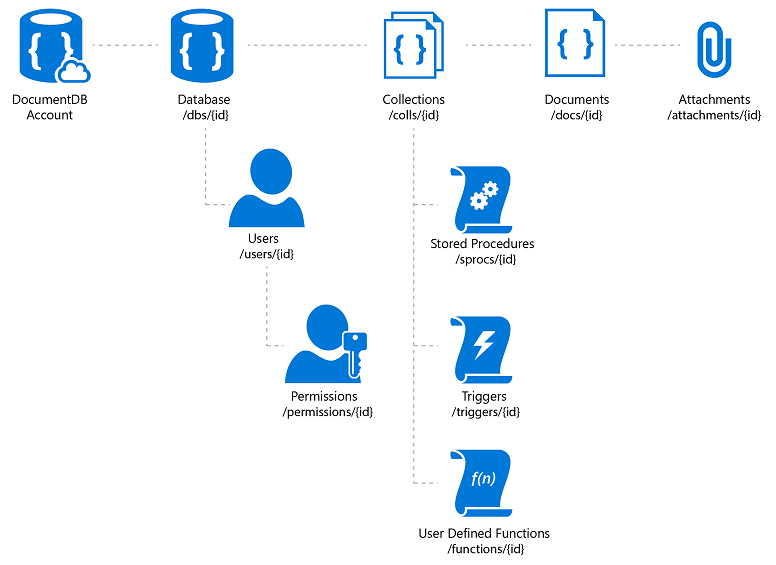
By reading this article, you’ll be able to answer the following questions:

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

## Hierarchical resource model

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

[AZURE.NOTE] DocumentDB offers a highly efficient TCP protocol which is also RESTful in its communication model, available through the [.NET client SDK](https://msdn.microsoft.com/library/azure/dn781482.aspx).

  
**Hierarchical resource model**

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

|  |  |
| --- | --- |
| Resource | Description |
| Database account | A database account is associated with a set of databases and a fixed amount of blob storage for attachments (preview feature). You can create one or more database accounts using your Azure subscription. For more information, visit our [pricing page](https://azure.microsoft.com/pricing/details/documentdb/). |
| 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](/documentation/articles/documentdb-performance-levels) 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. |
| Stored Procedure | Application logic written in JavaScript which is registered with a collection and transactionally executed within the database engine. |
| Trigger | Application logic written in JavaScript executed before or after either an insert, replace or delete operation. |
| UDF | Application logic written in JavaScript. UDFs enable you to model a custom query operator and thereby extend the core DocumentDB query language. |
| Document | User defined (arbitrary) JSON content. By default, no schema needs to be defined nor do secondary indices need to be provided for all the documents added to a collection. |
| (Preview) Attachment | An attachment is a special document containing references and associated metadata for external blob/media. The developer can choose to have the blob managed by DocumentDB or store it with an external blob service provider such as OneDrive, Dropbox, etc. |

## System vs. user defined resources

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

[AZURE.NOTE] Note that all system generated properties in a resource are prefixed with an underscore (\_) in their JSON representation.

<tbody>  
 <tr>  
 <td valign="top"><p><strong>Property</strong></p></td>  
 <td valign="top"><p><strong>User settable or system generated?</strong></p></td>  
 <td valign="top"><p><strong>Purpose</strong></p></td>  
 </tr>  
 <tr>  
 <td valign="top"><p>\_rid</p></td>  
 <td valign="top"><p>System generated</p></td>  
 <td valign="top"><p>System generated, unique and hierarchical identifier of the resource</p></td>  
 </tr>  
 <tr>  
 <td valign="top"><p>\_etag</p></td>  
 <td valign="top"><p>System generated</p></td>  
 <td valign="top"><p>etag of the resource required for optimistic concurrency control</p></td>  
 </tr>  
 <tr>  
 <td valign="top"><p>\_ts</p></td>  
 <td valign="top"><p>System generated</p></td>  
 <td valign="top"><p>Last updated timestamp of the resource</p></td>  
 </tr>  
 <tr>  
 <td valign="top"><p>\_self</p></td>  
 <td valign="top"><p>System generated</p></td>  
 <td valign="top"><p>Unique addressable URI of the resource</p></td>  
 </tr>  
 <tr>  
 <td valign="top"><p>id</p></td>  
 <td valign="top"><p>System generated</p></td>  
 <td valign="top"><p>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</p></td>  
 </tr>  
</tbody>

### Wire representation of resources

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

### Addressing a resource

All resources are URI addressable. The value of the \*\*\_self\*\* property of a resource represents the relative URI of the resource. The format of the URI consists of the /<feed>/{\_rid} path segments:

|Value of the \_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, the system will automatically generate a unique id for the document. The id is a user defined string, of up to 256 characters that is unique within the context of a specific parent resource.

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

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

## Database accounts

You can provision one or more DocumentDB database accounts using your Azure subscription. Each Standard tier database account will be given a minimum capacity of one S1 collection.

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

### Database account properties

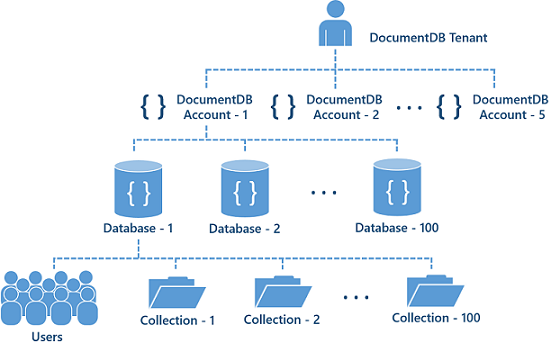
As part of provisioning and managing a database account you can configure and read the following properties:

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

Note that in addition to provisioning, configuring and managing your database account from the Azure Portal, you can also programmatically create and manage DocumentDB database accounts by using the [Azure DocumentDB REST APIs](https://msdn.microsoft.com/library/azure/dn781481.aspx) as well as [client SDKs](https://msdn.microsoft.com/library/azure/dn781482.aspx).

## Databases

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

  
**A Database is a logical container of users and collections**

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

### Elastic scale of a DocumentDB database

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

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

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

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

As with other resources in the DocumentDB resource model, databases can be created, replaced, deleted, read or enumerated easily using either [Azure DocumentDB REST APIs](https://msdn.microsoft.com/library/azure/dn781481.aspx) or any of the [client SDKs](https://msdn.microsoft.com/library/azure/dn781482.aspx). DocumentDB guarantees strong consistency for reading or querying the metadata of a database resource. Deleting a database automatically ensures that you cannot access any of the collections or users contained within it.

## Collections

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

### Elastic SSD backed document storage

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

### Automatic indexing of collections

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

### Configuring the indexing policy of a collection

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

* Choose whether the collection automatically indexes all of the documents or not. By default, all documents are automatically indexed. You can choose to turn off automatic indexing and selectively add only specific documents to the index. Conversely, you can selectively choose to exclude only specific documents. You can achieve this by setting the automatic property to be true or false on the indexingPolicy of a collection and using the [x-ms-indexingdirective] request header while inserting, replacing or deleting a document.
* Choose whether to include or exclude specific paths or patterns in your documents from the index. You can achieve this by setting includedPaths and excludedPaths on the indexingPolicy of a collection respectively. You can also configure the storage and performance trade-offs for range and hash queries for specific path patterns.
* Choose between synchronous (consistent) and asynchronous (lazy) index updates. By default, the index is updated synchronously on each insert, replace or delete of a document to the collection. This enables the queries to honor the same consistency level as that of the document reads. While DocumentDB is write optimized and supports sustained volumes of document writes along with synchronous index maintenance and serving consistent queries, you can configure certain collections to update their index lazily. Lazy indexing boosts the write performance further and is ideal for bulk ingestion scenarios for primarily read-heavy collections.

The indexing policy can be changed by executing a PUT on the collection. This can be achieved either through the [client SDK](https://msdn.microsoft.com/library/azure/dn781482.aspx), the [Azure Portal](https://portal.azure.com) or the [Azure DocumentDB REST APIs](https://msdn.microsoft.com/library/azure/dn781481.aspx).

### Querying a collection

The documents within a collection can have arbitrary schemas and you can query documents within a collection without providing any schema or secondary indices upfront. You can query the collection using the [DocumentDB SQL syntax](https://msdn.microsoft.com/library/azure/dn782250.aspx), which provides rich hierarchical, relational, and spatial operators and extensibility via JavaScript-based UDFs. JSON grammar allows for modeling JSON documents as trees with labels as the tree nodes. This is exploited both by DocumentDB’s automatic indexing techniques as well as DocumentDB’s SQL dialect. The DocumentDB query language consists of three main aspects:

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

The DocumentDB query model attempts to strike a balance between functionality, efficiency and simplicity. The DocumentDB database engine natively compiles and executes the SQL query statements. You can query a collection using the [Azure DocumentDB REST APIs](https://msdn.microsoft.com/library/azure/dn781481.aspx) or any of the [client SDKs](https://msdn.microsoft.com/library/azure/dn781482.aspx). The .NET SDK comes with a LINQ provider.

[AZURE.TIP] You can try out DocumentDB and run SQL queries against our dataset in the [Query Playground](https://www.documentdb.com/sql/demo).

### Multi-document transactions

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

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

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

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

The JavaScript logic registered at a collection level can then issue database operations on the documents of the given collection. DocumentDB implicitly wraps the JavaScript based stored procedures and triggers within an ambient ACID transactions with snapshot isolation across documents within a collection. During the course of its execution, if the JavaScript throws an exception, then the entire transaction is aborted. The resulting programming model is a very simple yet powerful. JavaScript developers get a “durable” programming model while still using their familiar language constructs and library primitives.

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

After creating a collection, you can register stored procedures, triggers and UDFs with a collection using the [Azure DocumentDB REST APIs](https://msdn.microsoft.com/library/azure/dn781481.aspx) or any of the [client SDKs](https://msdn.microsoft.com/library/azure/dn781482.aspx). After registration, you can reference and execute them. Consider the following stored procedure written entirely in JavaScript, the code below takes two arguments (book name and author name) and creates a new document, queries for a document and then updates it – all within an implicit ACID transaction. At any point during the execution, if a JavaScript exception is thrown, the entire transaction aborts.

function businessLogic(name, author) {  
 var context = getContext();  
 var collectionManager = context.getCollection();   
 var collectionLink = collectionManager.getSelfLink()  
   
 // create a new document.  
 collectionManager.createDocument(collectionLink,  
 {id: name, author: author},  
 function(err, documentCreated) {  
 if(err) throw new Error(err.message);  
   
 // filter documents by author  
 var filterQuery = "SELECT \* from root r WHERE r.author = 'George R.'";  
 collectionManager.queryDocuments(collectionLink,  
 filterQuery,  
 function(err, matchingDocuments) {  
 if(err) throw new Error(err.message);  
   
 context.getResponse().setBody(matchingDocuments.length);  
   
 // Replace the author name for all documents that satisfied the query.  
 for (var i = 0; i < matchingDocuments.length; i++) {  
 matchingDocuments[i].author = "George R. R. Martin";  
 // we don’t need to execute a callback because they are in parallel  
 collectionManager.replaceDocument(matchingDocuments[i].\_self,  
 matchingDocuments[i]);   
 }  
 })  
 })  
};

The client can “ship” the above JavaScript logic to the database for transactional execution via HTTP POST. For more information about using HTTP methods, see [RESTful interactions with DocumentDB resources](https://msdn.microsoft.com/library/azure/mt622086.aspx).

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

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

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

Collections in DocumentDB can be created, deleted, read or enumerated easily using either the [Azure DocumentDB REST APIs](https://msdn.microsoft.com/library/azure/dn781481.aspx) or any of the [client SDKs](https://msdn.microsoft.com/library/azure/dn781482.aspx). DocumentDB always provides strong consistency for reading or querying the metadata of a collection. Deleting a collection automatically ensures that you cannot access any of the documents, attachments, stored procedures, triggers, and UDFs contained within it.

## Stored procedures, triggers and User Defined Functions (UDF)

As described in the previous section, you can write application logic to run directly within a transaction inside of the database engine. The application logic can be written entirely in JavaScript and can be modeled as a stored procedure, trigger or a UDF. The JavaScript code within a stored procedure or a trigger can insert, replace, delete, read or query documents within a collection. On the other hand, the JavaScript within a UDF cannot insert, replace, or delete documents. UDFs enumerate the documents of a query’s result set and produce another result set. For multi-tenancy, DocumentDB enforces a strict reservation based resource governance. Each stored procedure, trigger or a UDF gets a fixed quantum of operating system resources to do its work. Furthermore, stored procedures, triggers or UDFs cannot link against external JavaScript libraries and are blacklisted if they exceed the resource budgets allocated to them. You can register, unregister stored procedures, triggers or UDFs with a collection by using the REST APIs. Upon registration a stored procedure, trigger, or a UDF is pre-compiled and stored as byte code which gets executed later. The following section illustrate how you can use the DocumentDB JavaScript SDK to register, execute, and unregister a stored procedure, trigger, and a UDF. The JavaScript SDK is a simple wrapper over the [DocumentDB REST APIs](https://msdn.microsoft.com/library/azure/dn781481.aspx).

### 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

Un-registering 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

Un-registering 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](https://msdn.microsoft.com/library/azure/dn782250.aspx).

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

Un-registering 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), un-registration (PUT), read/list (GET) and execution (POST) via the [DocumentDB JavaScript SDK](https://github.com/Azure/azure-documentdb-js), you can also use the [REST APIs](https://msdn.microsoft.com/library/azure/dn781481.aspx) or other [client SDKs](https://msdn.microsoft.com/library/azure/dn781482.aspx).

## Documents

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

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

As with all other resources, documents can be created, replaced, deleted, read, enumerated and queried easily using either REST APIs or any of the [client SDKs](https://msdn.microsoft.com/library/azure/dn781482.aspx). 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

[AZURE.NOTE] Attachment and media resources are preview features.

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

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

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

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

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

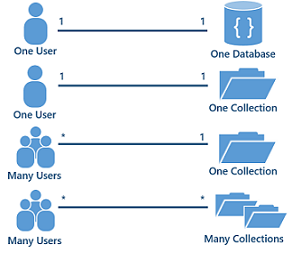
As with all other resources, attachments can be created, replaced, deleted, read or enumerated easily using either REST APIs or any of the client SDKs. As with documents, the read consistency level of attachments follows the consistency policy on the database account. This policy can be overridden on a per-request basis depending on data consistency requirements of your application. When querying for attachments, the read consistency follows the indexing mode set on the collection. For “consistent”, this follows the account’s consistency policy.   ## Users A DocumentDB user represents a logical namespace for grouping permissions. A DocumentDB user may correspond to a user in an identity management system or a predefined application role. For DocumentDB, a user simply represents an abstraction to group a set of permissions under a database.

For implementing multi-tenancy in your application, you can create users in DocumentDB which corresponds to your actual users or the tenants of your application. You can then create permissions for a given user that correspond to the access control over various collections, documents, attachments, etc.

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

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

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

  
**Sharding strategies and modeling users**

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

## Permissions

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

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

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

## Next steps

Learn more about working with resources by using HTTP commands in [RESTful interactions with DocumentDB resources](https://msdn.microsoft.com/library/azure/mt622086.aspx).