# DocumentDB server-side programming: Stored procedures, database triggers, and UDFs

Learn how DocumentDB’s language integrated, transactional execution of JavaScript lets developers write **stored procedures**, **triggers** and **user defined functions (UDFs)** natively in JavaScript. This allows you to write database program application logic that can be shipped and executed directly on the database storage partitions

We recommend getting started by watching the following video, where Andrew Liu provides a brief introduction to DocumentDB’s server-side database programming model.

[AZURE.VIDEO azure-demo-a-quick-intro-to-azure-documentdbs-server-side-javascript]

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 DocumentDB guarantee ACID?
* How do transactions work in DocumentDB?
* What are pre-triggers and post-triggers and how do I write one?
* How do I register and execute a stored procedure, trigger, or UDF in a RESTful manner by using HTTP?
* What DocumentDB SDKs are available to create and execute stored procedures, triggers, and UDFs?

## Introduction to Stored Procedure and UDF Programming

This approach of *“JavaScript as a modern day T-SQL”* frees application developers from the complexities of type system mismatches and object-relational mapping technologies. It also has a number of intrinsic advantages that can be utilized to build rich applications:

* **Procedural Logic:** JavaScript as a high level programming language, provides a rich and familiar interface to express business logic. You can perform complex sequences of operations closer to the data.
* **Atomic Transactions:** DocumentDB guarantees that database operations performed inside a single stored procedure or trigger are atomic. This lets an application combine related operations in a single batch so that either all of them succeed or none of them succeed.
* **Performance:** The fact that JSON is intrinsically mapped to the Javascript language type system and is also the basic unit of storage in DocumentDB allows for a number of optimizations like lazy materialization of JSON documents in the buffer pool and making them available on-demand to the executing code. There are more performance benefits associated with shipping business logic to the database:
  + Batching – Developers can group operations like inserts and submit them in bulk. The network traffic latency cost and the store overhead to create separate transactions are reduced significantly.
  + Pre-compilation – DocumentDB precompiles stored procedures, triggers and user defined functions (UDFs) to avoid JavaScript compilation cost for each invocation. The overhead of building the byte code for the procedural logic is amortized to a minimal value.
  + Sequencing – Many operations need a side-effect (“trigger”) that potentially involves doing one or many secondary store operations. Aside from atomicity, this is more performant when moved to the server.
* **Encapsulation:** Stored procedures can be used to group business logic in one place. This has two advantages:
  + It adds an abstraction layer on top of the raw data, which enables data architects to evolve their applications independently from the data. This is particularly advantageous when the data is schema-less, due to the brittle assumptions that may need to be baked into the application if they have to deal with data directly.
  + This abstraction lets enterprises keep their data secure by streamlining the access from the scripts.

The creation and execution of database triggers, stored procedure and custom query operators is supported through the [REST API](https://msdn.microsoft.com/library/azure/dn781481.aspx), [DocumentDB Studio](https://github.com/mingaliu/DocumentDBStudio/releases), and [client SDKs](/documentation/articles/documentdb-sdk-dotnet) in many platforms including .NET, Node.js and JavaScript.

**This tutorial uses the** [**Node.js SDK with Q Promises**](http://azure.github.io/azure-documentdb-node-q/) to illustrate syntax and usage of stored procedures, triggers, and UDFs.

## Stored procedures

### Example: Write a simple stored procedure

Let’s start with a simple stored procedure that returns a “Hello World” response.

var helloWorldStoredProc = {  
 id: "helloWorld",  
 body: function () {  
 var context = getContext();  
 var response = context.getResponse();  
  
 response.setBody("Hello, World");  
 }  
}

Stored procedures are registered per collection, and can operate on any document and attachment present in that collection. The following snippet shows how to register the helloWorld stored procedure with a collection.

// register the stored procedure  
var createdStoredProcedure;  
client.createStoredProcedureAsync('dbs/testdb/colls/testColl', helloWorldStoredProc)  
 .then(function (response) {  
 createdStoredProcedure = response.resource;  
 console.log("Successfully created stored procedure");  
 }, function (error) {  
 console.log("Error", error);  
 });

Once the stored procedure is registered, we can execute it against the collection, and read the results back at the client.

// execute the stored procedure  
client.executeStoredProcedureAsync('dbs/testdb/colls/testColl/sprocs/helloWorld')  
 .then(function (response) {  
 console.log(response.result); // "Hello, World"  
 }, function (err) {  
 console.log("Error", error);  
 });

The context object provides access to all operations that can be performed on DocumentDB storage, as well as access to the request and response objects. In this case, we used the response object to set the body of the response that was sent back to the client. For more details, refer to the [DocumentDB JavaScript server SDK documentation](http://azure.github.io/azure-documentdb-js-server/).

Let us expand on this example and add more database related functionality to the stored procedure. Stored procedures can create, update, read, query and delete documents and attachments inside the collection.

### Example: Write a stored procedure to create a document

The next snippet shows how to use the context object to interact with DocumentDB resources.

var createDocumentStoredProc = {  
 id: "createMyDocument",  
 body: function createMyDocument(documentToCreate) {  
 var context = getContext();  
 var collection = context.getCollection();  
  
 var accepted = collection.createDocument(collection.getSelfLink(),  
 documentToCreate,  
 function (err, documentCreated) {  
 if (err) throw new Error('Error' + err.message);  
 context.getResponse().setBody(documentCreated.id)  
 });  
 if (!accepted) return;  
 }  
}

This stored procedure takes as input documentToCreate, the body of a document to be created in the current collection. All such operations are asynchronous and depend on JavaScript function callbacks. The callback function has two parameters, one for the error object in case the operation fails, and one for the created object. Inside the callback, users can either handle the exception or throw an error. In case a callback is not provided and there is an error, the DocumentDB runtime throws an error.

In the example above, the callback throws an error if the operation failed. Otherwise, it sets the id of the created document as the body of the response to the client. Here is how this stored procedure is executed with input parameters.

// register the stored procedure  
client.createStoredProcedureAsync('dbs/testdb/colls/testColl', createDocumentStoredProc)  
 .then(function (response) {  
 var createdStoredProcedure = response.resource;  
  
 // run stored procedure to create a document  
 var docToCreate = {  
 id: "DocFromSproc",  
 book: "The Hitchhiker’s Guide to the Galaxy",  
 author: "Douglas Adams"  
 };  
  
 return client.executeStoredProcedureAsync('dbs/testdb/colls/testColl/sprocs/createMyDocument',  
 docToCreate);  
 }, function (error) {  
 console.log("Error", error);  
 })  
.then(function (response) {  
 console.log(response); // "DocFromSproc"  
}, function (error) {  
 console.log("Error", error);  
});

Note that this stored procedure can be modified to take an array of document bodies as input and create them all in the same stored procedure execution instead of multiple network requests to create each of them individually. This can be used to implement an efficient bulk importer for DocumentDB (discussed later in this tutorial).

The example described demonstrated how to use stored procedures. We will cover triggers and user defined functions (UDFs) later in the tutorial.

## Database program transactions

Transaction in a typical database can be defined as a sequence of operations performed as a single logical unit of work. Each transaction provides **ACID guarantees**. ACID is a well-known acronym that stands for four properties - Atomicity, Consistency, Isolation and Durability.

Briefly, atomicity guarantees that all the work done inside a transaction is treated as a single unit where either all of it is committed or none. Consistency makes sure that the data is always in a good internal state across transactions. Isolation guarantees that no two transactions interfere with each other – generally, most commercial systems provide multiple isolation levels that can be used based on the application needs. Durability ensures that any change that’s committed in the database will always be present.

In DocumentDB, JavaScript is hosted in the same memory space as the database. Hence, requests made within stored procedures and triggers execute in the same scope of a database session. This enables DocumentDB to guarantee ACID for all operations that are part of a single stored procedure/trigger. Consider the following stored procedure definition:

// JavaScript source code  
var exchangeItemsSproc = {  
 name: "exchangeItems",  
 body: function (playerId1, playerId2) {  
 var context = getContext();  
 var collection = context.getCollection();  
 var response = context.getResponse();  
  
 var player1Document, player2Document;  
  
 // query for players  
 var filterQuery = 'SELECT \* FROM Players p where p.id = "' + playerId1 + '"';  
 var accept = collection.queryDocuments(collection.getSelfLink(), filterQuery, {},  
 function (err, documents, responseOptions) {  
 if (err) throw new Error("Error" + err.message);  
  
 if (documents.length != 1) throw "Unable to find both names";  
 player1Document = documents[0];  
  
 var filterQuery2 = 'SELECT \* FROM Players p where p.id = "' + playerId2 + '"';  
 var accept2 = collection.queryDocuments(collection.getSelfLink(), filterQuery2, {},  
 function (err2, documents2, responseOptions2) {  
 if (err2) throw new Error("Error" + err2.message);  
 if (documents2.length != 1) throw "Unable to find both names";  
 player2Document = documents2[0];  
 swapItems(player1Document, player2Document);  
 return;  
 });  
 if (!accept2) throw "Unable to read player details, abort ";  
 });  
  
 if (!accept) throw "Unable to read player details, abort ";  
  
 // swap the two players’ items  
 function swapItems(player1, player2) {  
 var player1ItemSave = player1.item;  
 player1.item = player2.item;  
 player2.item = player1ItemSave;  
  
 var accept = collection.replaceDocument(player1.\_self, player1,  
 function (err, docReplaced) {  
 if (err) throw "Unable to update player 1, abort ";  
  
 var accept2 = collection.replaceDocument(player2.\_self, player2,  
 function (err2, docReplaced2) {  
 if (err) throw "Unable to update player 2, abort"  
 });  
  
 if (!accept2) throw "Unable to update player 2, abort";  
 });  
  
 if (!accept) throw "Unable to update player 1, abort";  
 }  
 }  
}  
  
// register the stored procedure in Node.js client  
client.createStoredProcedureAsync(collection.\_self, exchangeItemsSproc)  
 .then(function (response) {  
 var createdStoredProcedure = response.resource;  
 }  
);

This stored procedure uses transactions within a gaming app to trade items between two players in a single operation. The stored procedure attempts to read two documents each corresponding to the player IDs passed in as an argument. If both player documents are found, then the stored procedure updates the documents by swapping their items. If any errors are encountered along the way, it throws a JavaScript exception that implicitly aborts the transaction.

If the collection the stored procedure is registered against is a single-partition collection, then the transaction is scoped to all the docuemnts within the collection. If the collection is partitioned, then stored procedures are executed in the transaction scope of a single partition key. Each stored procedure execution must then include a partition key value corresponding to the scope the transaction must run under. For more details, see [DocumentDB Partitioning](/documentation/articles/documentdb-partition-data).

### Commit and rollback

Transactions are deeply and natively integrated into DocumentDB’s JavaScript programming model. Inside a JavaScript function, all operations are automatically wrapped under a single transaction. If the JavaScript completes without any exception, the operations to the database are committed. In effect, the “BEGIN TRANSACTION” and “COMMIT TRANSACTION” statements in relational databases are implicit in DocumentDB.

If there is any exception that’s propagated from the script, DocumentDB’s JavaScript runtime will roll back the whole transaction. As shown in the earlier example, throwing an exception is effectively equivalent to a “ROLLBACK TRANSACTION” in DocumentDB.

### Data consistency

Stored procedures and triggers are always executed on the primary replica of the DocumentDB collection. This ensures that reads from inside stored procedures offer strong consistency. Queries using user defined functions can be executed on the primary or any secondary replica, but we ensure to meet the requested consistency level by choosing the appropriate replica.

## Bounded execution

All DocumentDB operations must complete within the server specified request timeout duration. This constraint also applies to JavaScript functions (stored procedures, triggers and user-defined functions). If an operation does not complete with that time limit, the transaction is rolled back. JavaScript functions must finish within the time limit or implement a continuation based model to batch/resume execution.

In order to simplify development of stored procedures and triggers to handle time limits, all functions under the collection object (for create, read, replace, and delete of documents and attachments) return a Boolean value that represents whether that operation will complete. If this value is false, it is an indication that the time limit is about to expire and that the procedure must wrap up execution. Operations queued prior to the first unaccepted store operation are guaranteed to complete if the stored procedure completes in time and does not queue any more requests.

JavaScript functions are also bounded on resource consumption. DocumentDB reserves throughput per collection based on the provisioned size of a database account. Throughput is expressed in terms of a normalized unit of CPU, memory and IO consumption called request units or RUs. JavaScript functions can potentially use up a large number of RUs within a short time, and might get rate-limited if the collection’s limit is reached. Resource intensive stored procedures might also be quarantined to ensure availability of primitive database operations.

### Example: Bulk importing data into a database program

Below is an example of a stored procedure that is written to bulk-import documents into a collection. Note how the stored procedure handles bounded execution by checking the Boolean return value from createDocument, and then uses the count of documents inserted in each invocation of the stored procedure to track and resume progress across batches.

function bulkImport(docs) {  
 var collection = getContext().getCollection();  
 var collectionLink = collection.getSelfLink();  
  
 // The count of imported docs, also used as current doc index.  
 var count = 0;  
  
 // Validate input.  
 if (!docs) throw new Error("The array is undefined or null.");  
  
 var docsLength = docs.length;  
 if (docsLength == 0) {  
 getContext().getResponse().setBody(0);  
 }  
  
 // Call the create API to create a document.  
 tryCreate(docs[count], callback);  
  
 // Note that there are 2 exit conditions:  
 // 1) The createDocument request was not accepted.   
 // In this case the callback will not be called, we just call setBody and we are done.  
 // 2) The callback was called docs.length times.  
 // In this case all documents were created and we don’t need to call tryCreate anymore. Just call setBody and we are done.  
 function tryCreate(doc, callback) {  
 var isAccepted = collection.createDocument(collectionLink, doc, callback);  
  
 // If the request was accepted, callback will be called.  
 // Otherwise report current count back to the client,   
 // which will call the script again with remaining set of docs.  
 if (!isAccepted) getContext().getResponse().setBody(count);  
 }  
  
 // This is called when collection.createDocument is done in order to process the result.  
 function callback(err, doc, options) {  
 if (err) throw err;  
  
 // One more document has been inserted, increment the count.  
 count++;  
  
 if (count >= docsLength) {  
 // If we created all documents, we are done. Just set the response.  
 getContext().getResponse().setBody(count);  
 } else {  
 // Create next document.  
 tryCreate(docs[count], callback);  
 }  
 }  
}

## Database triggers

### Database pre-triggers

DocumentDB provides triggers that are executed or triggered by an operation on a document. For example, you can specify a pre-trigger when you are creating a document – this pre-trigger will run before the document is created. The following is an example of how pre-triggers can be used to validate the properties of a document that is being created:

var validateDocumentContentsTrigger = {  
 name: "validateDocumentContents",  
 body: function validate() {  
 var context = getContext();  
 var request = context.getRequest();  
  
 // document to be created in the current operation  
 var documentToCreate = request.getBody();  
  
 // validate properties  
 if (!("timestamp" in documentToCreate)) {  
 var ts = new Date();  
 documentToCreate["my timestamp"] = ts.getTime();  
 }  
  
 // update the document that will be created  
 request.setBody(documentToCreate);  
 },  
 triggerType: TriggerType.Pre,  
 triggerOperation: TriggerOperation.Create  
}

And the corresponding Node.js client-side registration code for the trigger:

// register pre-trigger  
client.createTriggerAsync(collection.self, validateDocumentContentsTrigger)  
 .then(function (response) {  
 console.log("Created", response.resource);  
 var docToCreate = {  
 id: "DocWithTrigger",  
 event: "Error",  
 source: "Network outage"  
 };  
  
 // run trigger while creating above document   
 var options = { preTriggerInclude: "validateDocumentContents" };  
  
 return client.createDocumentAsync(collection.self,  
 docToCreate, options);  
 }, function (error) {  
 console.log("Error", error);  
 })  
.then(function (response) {  
 console.log(response.resource); // document with timestamp property added  
}, function (error) {  
 console.log("Error", error);  
});

Pre-triggers cannot have any input parameters. The request object can be used to manipulate the request message associated with the operation. Here, the pre-trigger is being run with the creation of a document, and the request message body contains the document to be created in JSON format.

When triggers are registered, users can specify the operations that it can run with. This trigger was created with TriggerOperation.Create, which means the following is not permitted.

var options = { preTriggerInclude: "validateDocumentContents" };  
  
client.replaceDocumentAsync(docToReplace.self,  
 newDocBody, options)  
.then(function (response) {  
 console.log(response.resource);  
}, function (error) {  
 console.log("Error", error);  
});  
  
// Fails, can’t use a create trigger in a replace operation

### Database post-triggers

Post-triggers, like pre-triggers, are associated with an operation on a document and don’t take any input parameters. They run **after** the operation has completed, and have access to the response message that is sent to the client.

The following example shows post-triggers in action:

var updateMetadataTrigger = {  
 name: "updateMetadata",  
 body: function updateMetadata() {  
 var context = getContext();  
 var collection = context.getCollection();  
 var response = context.getResponse();  
  
 // document that was created  
 var createdDocument = response.getBody();  
  
 // query for metadata document  
 var filterQuery = 'SELECT \* FROM root r WHERE r.id = "\_metadata"';  
 var accept = collection.queryDocuments(collection.getSelfLink(), filterQuery,  
 updateMetadataCallback);  
 if(!accept) throw "Unable to update metadata, abort";  
   
 function updateMetadataCallback(err, documents, responseOptions) {  
 if(err) throw new Error("Error" + err.message);  
 if(documents.length != 1) throw 'Unable to find metadata document';  
   
 var metadataDocument = documents[0];  
   
 // update metadata  
 metadataDocument.createdDocuments += 1;  
 metadataDocument.createdNames += " " + createdDocument.id;  
 var accept = collection.replaceDocument(metadataDocument.\_self,  
 metadataDocument, function(err, docReplaced) {  
 if(err) throw "Unable to update metadata, abort";  
 });  
 if(!accept) throw "Unable to update metadata, abort";  
 return;   
 }   
 },  
 triggerType: TriggerType.Post,  
 triggerOperation: TriggerOperation.All  
}

The trigger can be registered as shown in the following sample.

// register post-trigger  
client.createTriggerAsync('dbs/testdb/colls/testColl', updateMetadataTrigger)  
 .then(function(createdTrigger) {   
 var docToCreate = {   
 name: "artist\_profile\_1023",  
 artist: "The Band",  
 albums: ["Hellujah", "Rotators", "Spinning Top"]  
 };  
  
 // run trigger while creating above document   
 var options = { postTriggerInclude: "updateMetadata" };  
   
 return client.createDocumentAsync(collection.self,  
 docToCreate, options);  
 }, function(error) {  
 console.log("Error" , error);  
 })  
.then(function(response) {  
 console.log(response.resource);   
}, function(error) {  
 console.log("Error" , error);  
});

This trigger queries for the metadata document and updates it with details about the newly created document.

One thing that is important to note is the **transactional** execution of triggers in DocumentDB. This post-trigger runs as part of the same transaction as the creation of the original document. Therefore, if we throw an exception from the post-trigger (say if we are unable to update the metadata document), the whole transaction will fail and be rolled back. No document will be created, and an exception will be returned.

## User-defined functions

User-defined functions (UDFs) are used to extend the DocumentDB SQL query language grammar and implement custom business logic. They can only be called from inside queries. They do not have access to the context object and are meant to be used as compute-only JavaScript. Therefore, UDFs can be run on secondary replicas of the DocumentDB service.

The following sample creates a UDF to calculate income tax based on rates for various income brackets, and then uses it inside a query to find all people who paid more than $20,000 in taxes.

var taxUdf = {  
 name: "tax",  
 body: function tax(income) {  
  
 if(income == undefined)   
 throw 'no input';  
  
 if (income < 1000)   
 return income \* 0.1;  
 else if (income < 10000)   
 return income \* 0.2;  
 else  
 return income \* 0.4;  
 }  
}

The UDF can subsequently be used in queries like in the following sample:

// register UDF  
client.createUserDefinedFunctionAsync('dbs/testdb/colls/testColl', taxUdf)  
 .then(function(response) {   
 console.log("Created", response.resource);  
  
 var query = 'SELECT \* FROM TaxPayers t WHERE udf.tax(t.income) > 20000';   
 return client.queryDocuments('dbs/testdb/colls/testColl',  
 query).toArrayAsync();  
 }, function(error) {  
 console.log("Error" , error);  
 })  
.then(function(response) {  
 var documents = response.feed;  
 console.log(response.resource);   
}, function(error) {  
 console.log("Error" , error);  
});

## JavaScript language-integrated query API

In addition to issuing queries using DocumentDB’s SQL grammar, the server-side SDK allows you to perform optimized queries using a fluent JavaScript interface without any knowledge of SQL. The JavaScript query API allows you to programmatically build queries by passing predicate functions into chainable function calls, with a syntax familiar to ECMAScript5’s Array built-ins and popular JavaScript libraries like lodash. Queries are parsed by the JavaScript runtime to be executed efficiently using DocumentDB’s indices.

[AZURE.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](http://azure.github.io/azure-documentdb-js-server/).

### 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

Details

Results in all documents (paginated with continuation token) as is.

Projects the id, message (aliased to msg), and action from all documents.

Queries for documents with the predicate: id = “X998\_Y998”.

Queries for documents that have a Tags property and Tags is an array containing the value 123.

Queries for documents with a predicate, id = “X998\_Y998”, and then projects the id and message (aliased to msg).

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](http://azure.github.io/azure-documentdb-js-server/) provides support for the most of the mainstream JavaScript language features as standardized by [ECMA-262](http://www.ecma-international.org/publications/standards/Ecma-262.htm).

### 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](/documentation/articles/documentdb-sdk-node) client, DocumentDB supports [.NET](/documentation/articles/documentdb-sdk-dotnet), [Java](/documentation/articles/documentdb-sdk-java), [JavaScript](http://azure.github.io/azure-documentdb-js/), and [Python SDKs](/documentation/articles/documentdb-sdk-python). 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](https://msdn.microsoft.com/library/azure/dn948556.aspx) to create a pre-trigger and create a document with the trigger enabled.

Trigger preTrigger = new Trigger()  
{  
 Id = "CapitalizeName",  
 Body = @"function() {  
 var item = getContext().getRequest().getBody();  
 item.id = item.id.toUpperCase();  
 getContext().getRequest().setBody(item);  
 }",  
 TriggerOperation = TriggerOperation.Create,  
 TriggerType = TriggerType.Pre  
};  
  
Document createdItem = await client.CreateDocumentAsync(UriFactory.CreateDocumentCollectionUri("db", "coll"), new Document { Id = "documentdb" },  
 new RequestOptions  
 {  
 PreTriggerInclude = new List<string> { "CapitalizeName" },  
 });

And the following example shows how to create a user defined function (UDF) and use it in a [DocumentDB SQL query](/documentation/articles/documentdb-sql-query).

UserDefinedFunction function = new UserDefinedFunction()  
{  
 Id = "LOWER",  
 Body = @"function(input)   
 {  
 return input.toLowerCase();  
 }"  
};  
  
foreach (Book book in client.CreateDocumentQuery(UriFactory.CreateDocumentCollectionUri("db", "coll"),  
 "SELECT \* FROM Books b WHERE udf.LOWER(b.Title) = 'war and peace'"))  
{  
 Console.WriteLine("Read {0} from query", book);  
}

## REST API

All DocumentDB operations can be performed in a RESTful manner. Stored procedures, triggers and user-defined functions can be registered under a collection by using HTTP POST. The following is an example of how to register a stored procedure:

POST https://<url>/sprocs/ HTTP/1.1  
authorization: <<auth>>  
x-ms-date: Thu, 07 Aug 2014 03:43:10 GMT  
  
  
var x = {  
 "name": "createAndAddProperty",  
 "body": function (docToCreate, addedPropertyName, addedPropertyValue) {  
 var collectionManager = getContext().getCollection();  
 collectionManager.createDocument(  
 collectionManager.getSelfLink(),  
 docToCreate,  
 function(err, docCreated) {  
 if(err) throw new Error('Error: ' + err.message);  
 docCreated[addedPropertyName] = addedPropertyValue;  
 getContext().getResponse().setBody(docCreated);  
 });  
 }  
}

The stored procedure is registered by executing a POST request against the URI 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](https://github.com/Azure/azure-documentdb-js-server/tree/master/samples/stored-procedures/bulkDelete.js), and [update](https://github.com/Azure/azure-documentdb-js-server/tree/master/samples/stored-procedures/update.js)) on our [Github repository](https://github.com/Azure/azure-documentdb-js-server/tree/master/samples).

Want to share your awesome stored procedure? Please, send us a pull-request!

## Next steps

Once you have one or more stored procedures, triggers, and user-defined functions created, you can load them and view them in the Azure Portal using Script Explorer. For more information, see [View stored procedures, triggers, and user-defined functions using the DocumentDB Script Explorer](/documentation/articles/documentdb-view-scripts).

You may also find the following references and resources useful in your path to learn more about DocumentDB server-side programming:

* [Azure DocumentDB SDKs](https://msdn.microsoft.com/library/azure/dn781482.aspx)
* [DocumentDB Studio](https://github.com/mingaliu/DocumentDBStudio/releases)
* [JSON](http://www.json.org/)
* [JavaScript ECMA-262](http://www.ecma-international.org/publications/standards/Ecma-262.htm)
* [JavaScript – JSON type system](http://www.json.org/js.html)
* [Secure and Portable Database Extensibility](http://dl.acm.org/citation.cfm?id=276339)
* [Service Oriented Database Architecture](http://dl.acm.org/citation.cfm?id=1066267&coll=Portal&dl=GUIDE)
* [Hosting the .NET Runtime in Microsoft SQL server](http://dl.acm.org/citation.cfm?id=1007669)