



# Connected System Plugin for Appian

Rob Munroe, Principal Solutions Architect, Appian Corporation

Version 1.0

Introduction	1
Audience	1
Compatibility	1
Installation	2
Plugin Design Principles	2
Special Considerations	2
Dates and Times	2
Object IDs	3
Binary	4
Connected System	6
Integrations	8
READ Integration Operations	9
List Databases	9
List Collections	10
Collection Find	11
Collection Count	13
Collection Aggregate	15
WRITE Integration Operations	17
Collection Find to JSON File	17
Collection Aggregate to JSON File	18
Create Collection	20
Create Index in Collection	21
Insert Many in Collection	22
Insert One in Collection	24
Update Many in Collection	25
Update One in Collection	27
Replace One in Collection	28
Delete Many in Collection	30
Delete One in Collection	31
Drop Collection	33
Common Configuration Parameters	34
Output Type	34
Database	34

Collection	35
Filter JSON	35
Output JSON As a Single Array	36
Save to Folder	36
Filename	36
Skip Automatic Date Time Conversion	37
Character Set	37
Collation	37
Read Preference	39
Read Concern	39
JSON Query Expression Functions	40
Top-Level Functions	41
M_query()	41
M_field()	41
Comparison Query Operators	42
M_eq()	42
M_gt()	42
M_gte()	42
M_in()	43
M_lt()	43
M_lte()	43
M_ne()	43
M_nin()	43
Logical Query Operators	44
M_and()	44
M_nor()	44
M_not()	45
M_or()	45
Element Query Operators	45
M_exists()	45
M_type()	45
Evaluation Query Operators	46
M_expr()	46

M_jsonSchema()	46
M_mod()	46
M_regex()	46
M_text()	47
M_where()	47
Query Operator Examples	48
Date and Time Examples	48
Changelog	50
Future Enhancements	50

# Introduction

This plugin implements an Appian [Connected System](#) for [MongoDB](#) using the [MongoDB Java Driver](#) in synchronous mode.

The benefits of using this plugin versus MongoDB's REST API include:

- Most common MongoDB operations are provided as low-code [Integration objects](#), making it very easy to quickly integrate Appian and MongoDB
- Uses connection pooling to reduce authentication time per operation
- Uses the [MongoDB Wire Protocol](#) with [BSON](#) instead of HTTP/REST with JSON for more efficient communication
- Automatically handles the conversion of MongoDB BSON Documents to Appian Dictionaries, or alternatively provides MongoDB-created JSON representations
- Can import from and export to Appian Documents containing JSON
- Configured using a single [Connection String](#), which is masked and encrypted

## Audience

Users of this Connected System plugin are expected to be familiar with the [core concepts of MongoDB](#), including topics such as querying and aggregation syntax. Additionally, it is expected that you are familiar with your own Databases, Collections, Document schemas, and MongoDB server infrastructure.

Users are also expected to be familiar with Appian, building Applications, and how and why [Connected Systems](#) and [Integration Objects](#) are used.

## Compatibility

This plugin was built and tested on Appian versions 19.4 and 20.2.

Version 1.0 of this plugin uses the [MongoDB Java Driver version 3.12.5](#) and should be compatible with any version of MongoDB from version 2.6 to 4.2 (the latest as of this writing). See this [compatibility chart](#) for full details.

This plugin was testing against a [MongoDB Atlas](#) a 3-node Replica Set instance running version 4.2.8 as well as a Standalone MongoDB 4.2.8 Community instance running on Ubuntu 20.04.

It is expected that your MongoDB instance(s) allow network connections from your Appian instance(s).

## Installation

If installing to a fully-managed Appian Cloud instance, install using the [Plugins](#) panel of the [Administration Console](#).

If installing to a self-managed Appian instance, copy the `ps-plugin-MongoDbConnectedSystem-X.X.jar` file to the `<APPIAN_HOME>/_admin/plugins` directory.

## Plugin Design Principles

This plugin was designed to mirror the functionality provided by MongoDB's Java Driver. We have implemented the most common functionality as individual Integration Operations and are striving for 100% feature completeness over time. Please let us know on [Appian Community](#) if there are missing features that you require.

Many of the operations of the MongoDB Java Driver take as arguments MongoDB BSON Documents. As such, Integration Operations that require BSON Documents will instead accept JSON strings, which the plugin handles converting to BSON. We have included a full suite of [JSON Query Expression Functions](#) to cleanly and easily generate MongoDB JSON-based queries.

## Special Considerations

Due to how the MongoDB Java Driver returns Object IDs and Binary data types, special handling must be performed by the Connected System as described below.

**It is critical that you understand these special cases and design for them accordingly**

## Dates and Times

Dates and times in MongoDB are different from most other databases in that it stores them in UTC by default and will convert any local time representations into this form. Applications that must operate or report on some unmodified local time value must store the time zone offset alongside the UTC timestamp (e.g. as a separate field) and compute the original local time in their application logic.

It is up to you and your Application to account for this.

When storing Date and Time values from Appian to MongoDB, MongoDB will convert them to UTC. Therefore, any of your date queries should be against UTC dates and times. Use Appian's built-in Date and Time [functions](#) to ensure the dates being sent to MongoDB make sense in this regard.

Also note that unlike Appian, there is no date *without time* data type in MongoDB. When wishing to store a date without time, use midnight UTC of that date (e.g. `2020-07-01T00:00:00.000Z`) and query accordingly. The [JSON Query Expression Functions](#) will take this into account and convert Appian Dates to this format.

## Object IDs

While the [MongoDB Object ID](#) data type is most often represented as a string (e.g. `"5efa0b06fc13ae730e00024a"`), it is stored internally as 12-byte values broken down into several data points. It is far easier to work with the string value in Appian, so this Connected System will return a sub-Dictionary of the below form for each Object ID in the resulting dataset.

**Note:** This transformation only applies to results returned as Appian Dictionaries. JSON results use the MongoDB JSON notation.

A MongoDB Document representing this value:

```
{
  _id: ObjectId("5efa0b06fc13ae730e00024a")
  ...
}
```

Would be returned as an Appian Dictionary like this:

```
{
  _id: {
    oid: "5efa0b06fc13ae730e00024a"
  }
  ...
}
```

The key detail here is that Object ID fields will be accessed like this in Appian:

```
local!theObjectId: local!myDocument._id.oid
```

## Object ID CDT

This plugin contains a CDT named `{urn:com:appian:types:MongoDB}ObjectId` that can be used to represent these values in a consistent manner. MongoDB Document properties in Dictionaries can be cast directly to this CDT.

It is highly recommended that you use this CDT when creating your own CDTs that represent the MongoDB Documents used in your application. Using it also helps convert Appian Dictionaries representing MongoDB Documents to Mongo-friendly JSON using the `mdb_tojson` function, and the `mdb_tojson` function will return this value as well.

See the `{urn:com:appian:types:MCSO}MCSO_Customer` CDT in the demo application for example.

## Binary

MongoDB's Binary data type allows you to store chunks of binary data in a MongoDB Document, however Appian does not support storing binary data in Dictionaries. To work around this any binary results will be returned as Base64 encoded text.

**Note:** This transformation only applies to results returned as Appian Dictionaries. JSON results use the MongoDB JSON notation.

**Warning:** Returning large amounts of Base64 encoded binary data to Appian can have severe impacts on the performance of the Appian environment. A best practice would be to use a projection and eliminate the binary field from the MongoDB Document.

A MongoDB Document representing this value:

```
{
  binaryField: Binary("... Binary data value ...", 0)
  ...
}
```

Would be returned as an Appian Dictionary like this:

```
{
  binaryField: {
    binary: "...Base64 encoded data...",
    type: "0"
  }
  ...
}
```



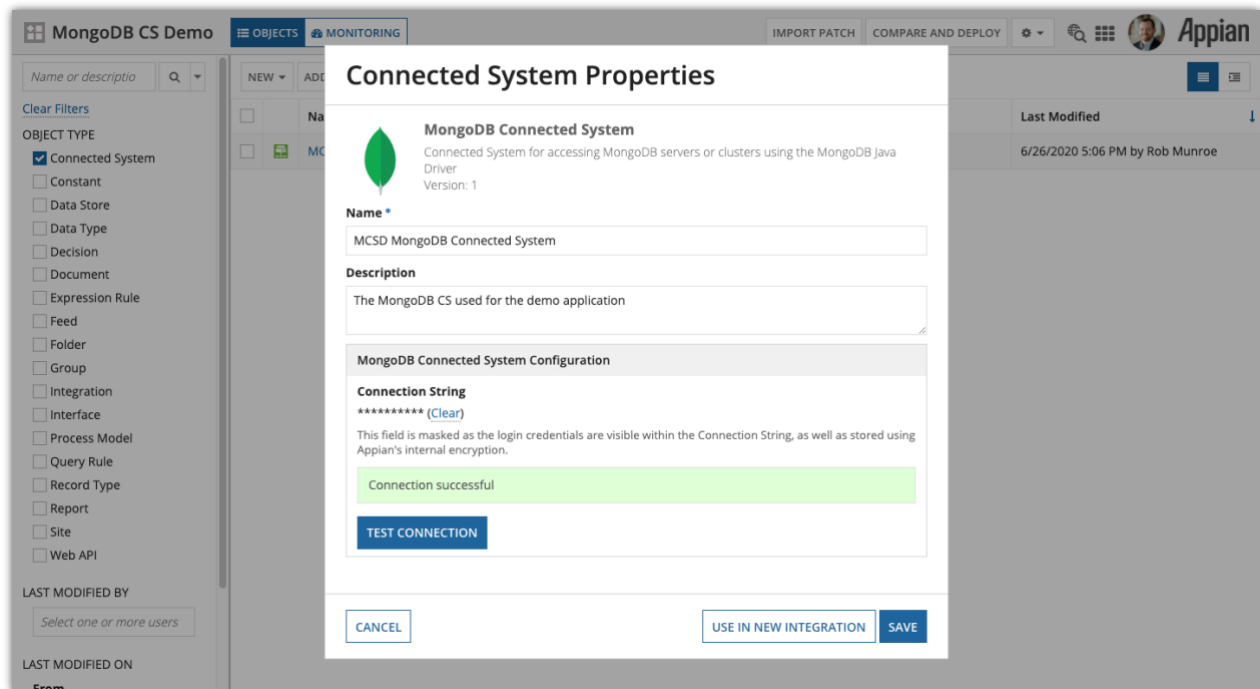
## Binary CDT

Similarly to Object ID, this plugin contains a CDT named `{urn:com:appian:types:MongoDB}Binary` that can be used to represent these values in a consistent manner. MongoDB Document properties in Dictionaries can be cast directly to this CDT.

It is highly recommended that you use this CDT when creating your own CDTs that represent the MongoDB Documents used in your application. Using it also helps convert Appian Dictionaries representing MongoDB Documents to Mongo-friendly JSON using the `mdb_tojson` function, and the `mdb_tojson` function will return this value as well.

See the `{urn:com:appian:types:MCSD}MCSD_Customer` CDT in the demo application for example.

# Connected System

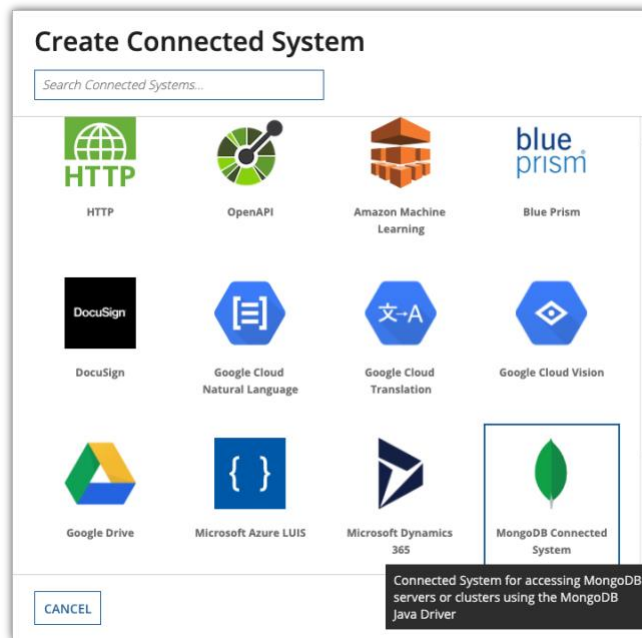


This plugin provides a single Connected System Template for connecting to MongoDB named **MongoDB Connected System**. Configuration of the Connected System is very simple and only requires the [MongoDB Connection String](#). It is up to you to fully understand the MongoDB Connection String format, as many parameters can be set that affect the functionality of the [MongoDB Java Driver](#) used by this plugin. A good rule of thumb is if you can connect to and work with your MongoDB Database using your Connection String in the [MongoDB Compass](#) application, you should have no issue using that Connection String with this Connected System.

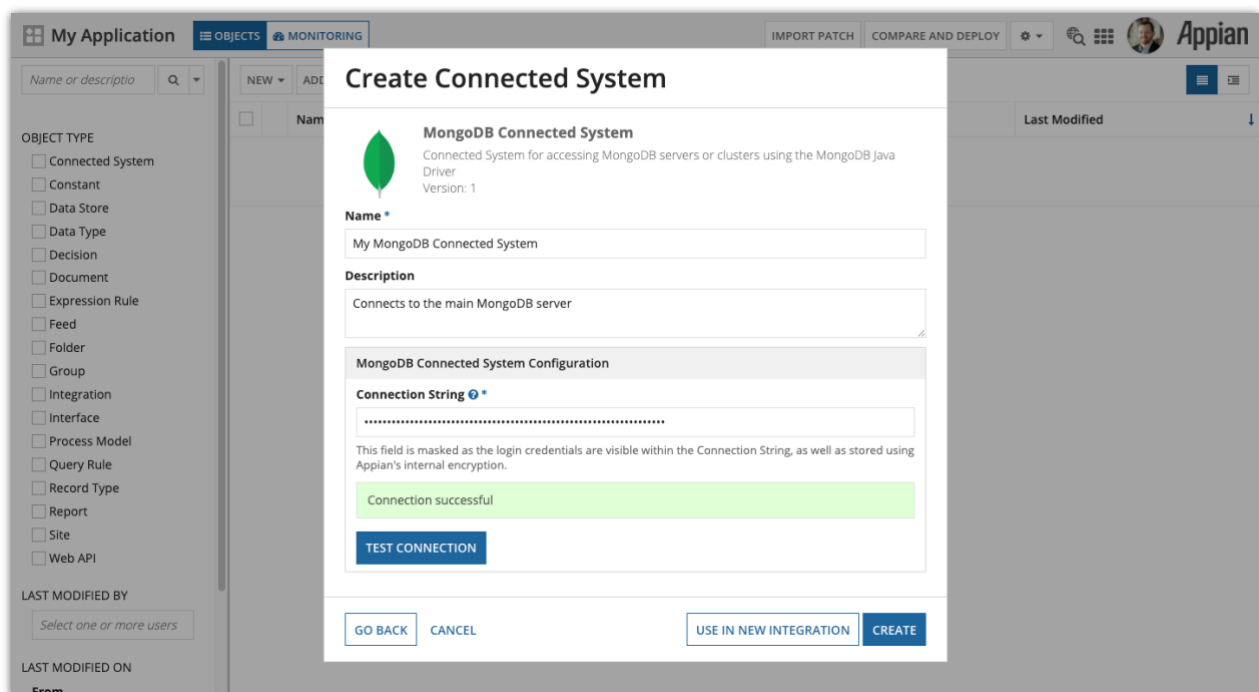
The first step in using this plugin is to ensure it has been installed in your Appian environment. Please refer to the [Appian Documentation for installing plugins](#).

To add a new MongoDB Connected System in your Application, open the application in Designer, and click **New > Connected System**.

Select the **MongoDB Connected System**



Give it a **Name** and **Description**, and then paste your MongoDB **Connection String** in the provided text box and click **Test Connection**. Assuming you see “*Connection successful*” then simply click the **Create** button to finish.



# Integrations


Creating an Integration will be the same steps for all Integration operations listed below. Start by clicking **New > Integration** in your application in Designer.

Enter the name of your MongoDB **Connected System** object, select the Integration **Operation**, then give it a **Name** and **Description** and the folder to **Save In** and click **Create**.

## Create Integration

☒ Use a connected system ☐ Create from scratch (HTTP only) ☐ Duplicate existing integration

**Connected System \***

 My MongoDB Connected System (MongoDB Connected System) ✕

**Operation \***

Collection Find ▼

Performs the Find operation in the given MongoDB Collection



**Name \***

MyApp\_findCustomersInZipCode

**Description**

Takes in a US zip code and returns all customers with an address in that zip.

**Save In \***

 My Application ✕ 

Create New Rule Folder

CANCEL

CREATE

Each Integration Operation will be configured differently, as noted below.

**Note:** Unless noted otherwise, all parameters in all Integration Operations are expressionable, meaning that they can be mapped to rule inputs or otherwise derived at runtime.

# READ Integration Operations

This section details all Integration Operations supported by the Connected System in a **READ** context. They can be used anywhere that Expressions are evaluated.

## List Databases

This Integration executes the [List Databases](#) operation to list all Databases available in the Connected System instance.

The screenshot displays the Appian configuration interface for the 'MCSD\_Integration\_listDatabases' operation. On the left, the 'Connected System' is set to 'MCSD MongoDB Connected System'. The 'Operation' is 'List Databases', with a description 'Get the list of available MongoDB Databases'. The 'Output Type' is set to 'Dictionary'. A 'TEST REQUEST' button is at the bottom. The right pane shows the 'Result' tab with a green 'Success!' banner. Below this, performance metrics are shown: 'Time: 32 ms', 'Prepare: 1 ms', 'Execute: 29 ms (Send/Wait/Receive: 5 ms)', and 'Transform: 2 ms'. The 'Value' section shows a nested dictionary structure: 'success: true (Boolean)', 'result: Dictionary', and 'databases: List of Dictionary - 4 items'. The 'databases' list contains four entries, each a dictionary with 'name' (Text), 'sizeOnDisk' (Number (Decimal)), and 'empty' (Boolean) fields. The entries are: 'admin' (size 184320), 'appian' (size 1208320), 'config' (size 94208), and 'local' (size 73728). The 'error' field is 'null (Null)'.

The `result.databases.name` property is the MongoDB Database name that can be used as any Integration Operation's [Database](#) parameter value.

## Parameters

### Output Type

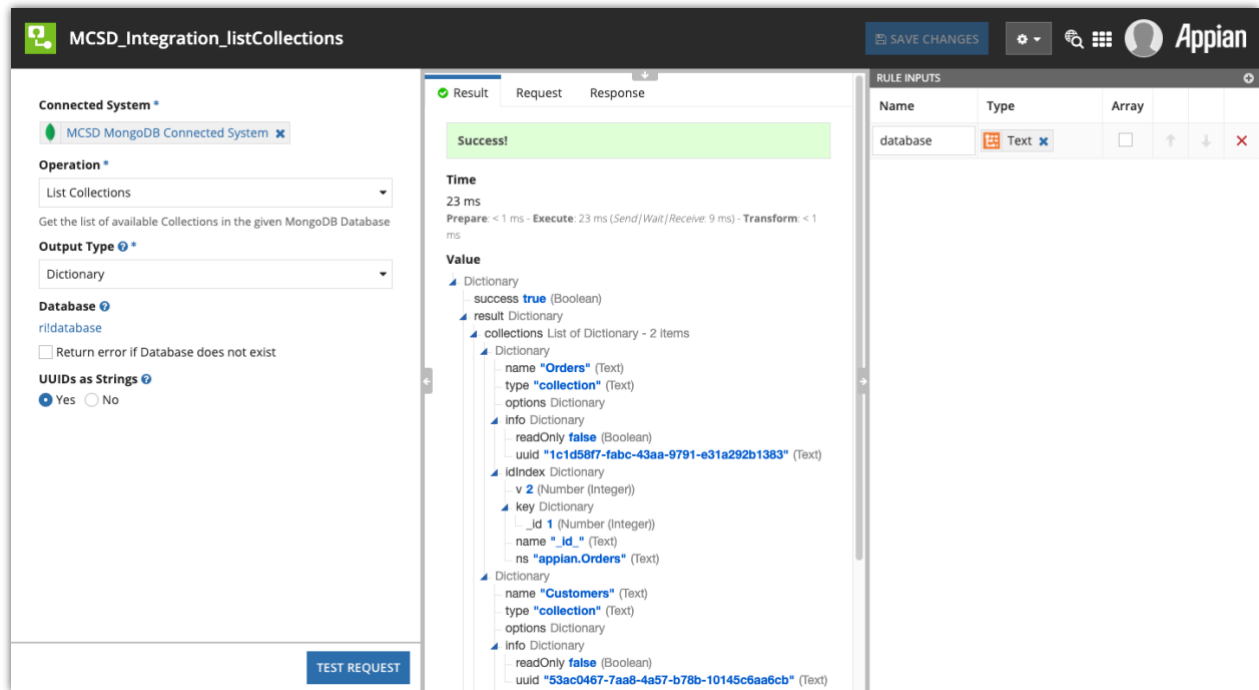
Refer to [Output Type](#) in Common Configuration Parameters.

### Result Value

Field Name	Data Type	Description
<code>databases</code>	List of Dictionary	Dictionaries represent the <a href="#">List Databases output data</a>

# List Collections

This Integration executes the [List Collections](#) operation to list all MongoDB Collections in the given Database available to the connected user.



## Parameters

### Output Type

Refer to [Output Type](#) in Common Configuration Parameters.

### Database

Refer to [Database](#) in Common Configuration Parameters.

### UUIDs as String

MongoDB stores UUIDs as a special data type internally. When returned to Appian, they are sub-dictionaries with several properties. Chances are this is not as useful to you, so setting Yes here will ensure the string representation of the UUID is returned instead of the sub-dictionary.

## Result Value

Field Name	Data Type	Description
collections	List of Dictionary	Dictionaries represent the <a href="#">List Collections output data</a>
database	Text	The database used

# Collection Find

Performs the [Find](#) operation in the given MongoDB Collection and returns the matching MongoDB Documents. This is the standard operation for querying MongoDB Documents.

The screenshot displays the Appian interface for the **MCSD\_Integration\_collectionFind** operation. The left panel shows configuration options: **Operation \*** is set to 'Collection Find', **Output Type \*** is 'Dictionary', **Database** is 'constMCSD\_MDB\_DATABASE', and **Collection** is 'r!collectionName'. A **Filter JSON** editor contains a query: 

```
1 M_query(
2   M_and(
3     M_field(
4       "lastName",
5       M_in(
6         "McIlraith",
7         "Orcott",
8         "Toulch",
9         "Jemmett"
10      )
11    ),
12    M_field("disabled", M_ne(true))
13  )
14 )
15
```

 The right panel shows the **Result** tab with a 'Success!' message. It details the execution time (200 ms) and the response structure, which includes a dictionary with fields like 'success', 'result', 'database', 'collection', 'documents', '\_id', 'old', 'createdOn', 'createdBy', 'firstName', 'lastName', 'emailAddress', 'phoneHome', 'phoneWork', 'address', 'loc', 'type', 'coordinates', 'number', 'street', and 'unit'.

## Parameters

### Output Type

Refer to [Output Type](#) in Common Configuration Parameters.

### Database

Refer to [Database](#) in Common Configuration Parameters.

### Collection

Refer to [Collection](#) in Common Configuration Parameters.

### Filter JSON

Refer to [Filter JSON](#) in Common Configuration Parameters.

## Sort JSON

A JSON string representing the sort order for a `Collection.Find()` query. Sort specifies the order in which the query returns matching documents. Example, sorting by last name ascendingly, then first name ascendingly:

```
{ "lastName": 1, "firstName": 1 }
```

## Projection JSON

A JSON string representing a [Projection](#) for a `Collection.Find()` query. Projections limit the amount of data that MongoDB returns.

Example, returning MongoDB Documents that only contain first name, last name, and postal code, and omits the `_id` (which is always projected unless omitted):

```
{ "firstName": 1, "lastName": 1, "address.postalCode": 1, "_id": 0 }
```

## Limit

Sets the number of MongoDB Document results to return. Useful for mapping `a!pagingInfo.batchSize` to use paging in your queries.

## Skip

Sets the number of MongoDB Document results to skip before returning. Useful for mapping `a!pagingInfo.startIndex` to use paging in your queries.

## Collation

Refer to [Collation](#) in Common Configuration Parameters.

## Max Processing Time

Specifies a cumulative time limit in milliseconds for **processing** operations on a Find operation. Note that this is not the complete time to perform the Integration Operation, nor the entire query on the MongoDB server, but only the time MongoDB is processing the query.

## Read Preference

Refer to [Read Preference](#) in Common Configuration Parameters.

## Read Concern

Refer to [Read Concern](#) in Common Configuration Parameters.



## Include Record Id

Modifies the output of a query by adding a field `recordId` to matching MongoDB Documents. Record Id is the internal key which uniquely identifies a MongoDB Document in a Collection. Note, this is different from a MongoDB Document's Object Id.

## Result Value

Field Name	Data Type	Description
database	Text	The database used
collection	Text	The collection used
documents	List of Dictionary OR List of Text	The MongoDB Documents matched by the Filter JSON query, either as Dictionaries or JSON strings depending on the Output Type selected

## Collection Count

Performs the `Count` operation on the Collection, returning the number of MongoDB Documents that match the provided Filter JSON.

The screenshot displays the Appian configuration for the 'Collection Count' operation. On the left, the 'Connected System' is 'MCSDB MongoDB Connected System'. The 'Operation' is set to 'Collection Count'. The 'Database' is 'rtdatabaseName' and the 'Collection' is 'rtcollectionName'. The 'Filter JSON' is defined as:

```
1 m_query(  
2   m_field(  
3     "lastName",  
4     m_regex("^St", "")  
5   )  
6 )
```

On the right, the 'Rule Input Name' table shows the configuration for the database and collection. Below this, the 'Result' tab shows a successful outcome with the following details:

- Time:** 521 ms (Prepare: < 1 ms, Execute: 521 ms, Transform: < 1 ms)
- Value:** Dictionary containing:
  - success: true (Boolean)
  - result: Dictionary containing:
    - count: 16 (Number (Decimal))
    - database: "applan" (Text)
    - collection: "Customers" (Text)
    - error: null (Null)

This is useful for determining how many total results match a given Filter JSON without returning the data. Using this in conjunction with `Collection Find` and `!pagingInfo()` allows for complete paging of your queries.

## Parameters

### Database

Refer to [Database](#) in Common Configuration Parameters.

### Collection

Refer to [Collection](#) in Common Configuration Parameters.

### Filter JSON

Refer to [Filter JSON](#) in Common Configuration Parameters.

### Collation

Refer to [Collation](#) in Common Configuration Parameters.

### Read Preference

Refer to [Read Preference](#) in Common Configuration Parameters.

### Read Concern

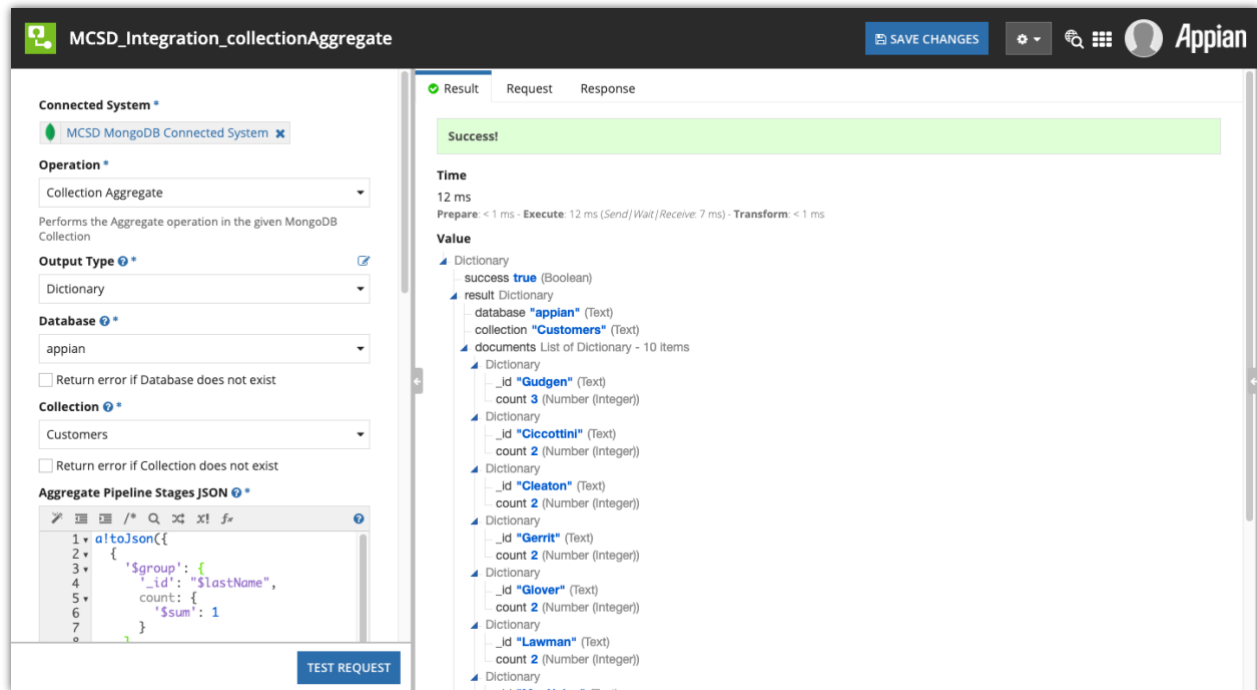
Refer to [Read Concern](#) in Common Configuration Parameters.

## Result Value

Field Name	Data Type	Description
database	Text	The database used
collection	Text	The collection used
count	Number (Integer)	The number of MongoDB Documents matched by the Filter JSON

# Collection Aggregate

Performs the [Aggregate](#) operation, taking in an [Aggregation Pipeline](#) in the form of a single Text parameter that represents an array of JSON pipeline operations.



Aggregates are MongoDB's method of performing analytic operations on Collections, allowing for operations such as `$match` and `$group`. Please see the MongoDB documentation for further details.

## Parameters

### Output Type

Refer to [Output Type](#) in Common Configuration Parameters.

### Database

Refer to [Database](#) in Common Configuration Parameters.

### Collection

Refer to [Collection](#) in Common Configuration Parameters.

### Aggregate Pipeline Stages JSON

This parameter allows you to provide a collection of “stages” of an Aggregate Pipeline in the form of a single Text value that represents an array of JSON pipeline operations. For example,

this JSON value represents an Aggregate Pipeline that returns the top 10 most common last names from a collection (e.g. Customers).

```
[
  {
    "$group": {
      "_id": "$lastName",
      "count": {
        "$sum": 1
      }
    }
  },
  {
    "$sort": {
      "count": -1,
      "_id": 1
    }
  },
  {
    "$limit": 10
  }
]
```

## Collation

Refer to [Collation](#) in Common Configuration Parameters.

## Read Preference

Refer to [Read Preference](#) in Common Configuration Parameters.

## Read Concern

Refer to [Read Concern](#) in Common Configuration Parameters.

## Result Value

Field Name	Data Type	Description
database	Text	The database used
collection	Text	The collection used
documents	List of Dictionary OR List of Text	The MongoDB Documents produced by the Aggregate operation, either as Dictionaries or JSON strings depending on the Output Type selected

# WRITE Integration Operations

This section details all Integration Operations supported by the Connected System in a **WRITE** context. These Integrations can only be used in the [Call Integration Smart Service](#), in a [Web API](#) that uses a **POST**, **PUT**, or **DELETE** Request, or in a SAIL [Save Into](#) event.

## Collection Find to JSON File

Performs the [Find](#) operation in the given MongoDB Collection and exports the results as JSON to an Appian Document. This is identical to the READ [Collection Find](#) operation except that a document is created from the output.

The screenshot shows the Appian integration configuration interface for 'MyApp\_exportToFile'. The interface is divided into two main sections: configuration on the left and execution results on the right.

**Configuration Section (Left):**

- Connected System:** My MongoDB Connected System
- Operation:** Collection Find to JSON File
- Output JSON As a Single Array:** Yes (selected)
- Save to Folder:** Export Files
- Filename:** now() & ".json"
- Character Set:** UTF-8
- Database:** appian
- Collection:** (empty)

**Execution Results Section (Right):**

- Status:** Success!
- Time:** 376 ms
- Value: Result:**
  - database "appian" (Text)
  - jsonDocument 1334 - 2020 2:24 PM GMT+00:00.json (Document)
  - collection "Customers" (Text)

## Parameters

**Note:** Only the parameters that differ from the READ version of [Collection Find](#) above are discussed here. For all others, see [above](#).

### Output JSON As a Single Array

Refer to [Output JSON As a Single Array](#) in Common Configuration Parameters.

### Save to Folder

Refer to [Save to Folder](#) in Common Configuration Parameters.

## Filename

Refer to [Filename](#) in Common Configuration Parameters.

## Character Set

Refer to [Character Set](#) in Common Configuration Parameters.

## Result Value

Field Name	Data Type	Description
database	Text	The database used
collection	Text	The collection used
jsonDocument	Appian Document	The output file in Appian's content management

## Collection Aggregate to JSON File

Performs the Aggregate operation in the given MongoDB Collection and exports the results as JSON to an Appian Document. This is identical to the READ [Collection Aggregate](#) operation except that a document is created from the output.

The screenshot displays the Appian configuration interface for the 'Collection Aggregate to JSON File' operation. The left pane shows the configuration details, and the right pane shows the successful execution result.

**Configuration Details (Left Pane):**

- Connected System:** MCSD MongoDB Connected System
- Operation:** Collection Aggregate to JSON File
- Output JSON As a Single Array:** Yes (selected)
- Save to Folder:** Export Files
- Filename:** topLastNames.json
- Character Set:** UTF-8
- Database:** appian
- ☐ Return error if Database does not exist

**Execution Result (Right Pane):**

- Status:** Success!
- Time:** 212 ms
- Prepare:** 1 ms - **Execute:** 211 ms (Send/Wait/Receive: 157 ms) - **Transform:** < 1 ms
- Value: Result**
  - Dictionary
    - database "appian" (Text)
    - jsonDocument 1338 - topLastNames.json (Document)
    - collection "Customers" (Text)

## Parameters

**Note:** Only the parameters that differ from the READ version of [Collection Aggregate](#) above are discussed here. For all others, see [above](#).

### Output JSON As a Single Array

Refer to [Output JSON As a Single Array](#) in Common Configuration Parameters.

### Save to Folder

Refer to [Save to Folder](#) in Common Configuration Parameters.

### Filename

Refer to [Filename](#) in Common Configuration Parameters.

### Character Set

Refer to [Character Set](#) in Common Configuration Parameters.

## Result Value

Field Name	Data Type	Description
database	Text	The database used
collection	Text	The collection used
jsonDocument	Appian Document	The output file in Appian's content management

# Create Collection

Performs the [Create Collection](#) operation, explicitly creating a new Collection in the given Database.

The screenshot displays the Appian interface for the 'MCSG\_Integration\_createCollection' process. The left panel contains configuration fields: 'Connected System' is set to 'MCSG MongoDB Connected System'; 'Operation' is 'Create Collection'; 'Database' is 'admin'; and 'New Collection Name' is 'Categories'. A checkbox for 'Return error if Database does not exist' is unchecked. The right panel shows the 'Result' tab with a green 'Success!' message. Below this, it lists the execution time as 21 ms and provides a detailed 'Value: Result' dictionary: 'collectionCreated' is true (Boolean), 'database' is 'admin' (Text), and 'collection' is 'Categories' (Text). A 'TEST REQUEST' button is located at the bottom of the configuration panel.

**Note:** It is possible to implicitly create a Collection using [Insert One](#) or [Insert Many](#) by specifying a **Collection** name that does not exist and ensuring that the **Return error if Collection does not exist** checkbox is not checked.

## Parameters

### Database

Refer to [Database](#) in Common Configuration Parameters.

### Collection Name

The name of the new Collection to be created.



## Result Value

Field Name	Data Type	Description
database	Text	The database used
collection	Text	The collection name provided
collectionCreated	Boolean	Whether the Collection was created or not

## Create Index in Collection

Performs the [Create Index](#) operation, adding a new index in the given Collection. [Indexes](#) are critical for ensuring good performance on any operation that includes a Filter JSON value.

The screenshot displays the Appian configuration interface for the 'Create Index in Collection' operation. The left pane shows the configuration details:

- Connected System:** MCSDB MongoDB Connected System
- Operation:** Create Index in Collection
- Database:** cons!MCSDB\_MDB\_DATABASE
- Collection:** cons!MCSDB\_MDB\_COLLECTION\_CUSTOMERS
- Index JSON:**

```
1 { "address.loc" : "2dsphere" }
```

The right pane shows the execution result under the 'Result' tab:

- Status:** Success!
- Time:** 511 ms
- Prepare:** < 1 ms - **Execute:** 511 ms (Send/Wait/Receive 25 ms) - **Transform:** < 1 ms
- Value: Result:** Dictionary
  - database "applan" (Text)
  - collection "Customers" (Text)
  - indexName "address.loc\_2dsphere" (Text)

## Parameters

### Database

Refer to [Database](#) in Common Configuration Parameters.

### Collection

Refer to [Collection](#) in Common Configuration Parameters.

## Index JSON

The [MongoDB Index Document](#) in JSON form to instruct MongoDB how to create the new index.

## Result Value

Field Name	Data Type	Description
database	Text	The database used
collection	Text	The collection used
indexName	Text	The name of the newly created Index

## Insert Many in Collection

Performs the [Insert Many](#) operation, creating new MongoDB Document instances in the Collection provided.

The screenshot shows the 'Insert Many in Collection' operation configuration in Appian. The left sidebar contains settings for the operation, including the database 'cons!MCSDB\_DATABASE', collection 'ri!collectionName', and JSON source 'JSON String'. The main area displays a rule input table with columns 'Rule Input Name', 'Expression', and 'Value'. The 'collectionName' input is linked to the 'Categories' value. The 'jsonArray' input is linked to an expression that uses the 'altoJson' function to convert a list of text strings into a JSON array. The bottom right shows the 'Result' tab with a 'Success!' message and a 'Value: Result' section displaying a dictionary with document count, database, and collection details.

Rule Input Name	Expression	Value
collectionName (Text)	1	Categories
jsonArray (List of Text String)	<pre>1 + { 2 +   altoJson({ 3 +     category: "Sales", 4 +     employeeCount: 285 5 +   }) 6 + }</pre>	List of Text String: 2 Items {"category":"Sales","employeeCount":285} {"category":"Information Technology","e...More

**Result**  
Success!

**Time**  
76 ms  
Prepare: < 1 ms - Execute: 76 ms (Send/Wait/Receive: 64 ms) - Transform: < 1 ms

**Value: Result**  
Dictionary  
- documentCount 2 (Number (Integer))  
- database "applan" (Text)  
- collection "Categories" (Text)

This Integration Operation allows you to select the source of the JSON to be inserted, either as a List of Text (an array of JSON documents) or by reading JSON from an Appian Document.

When used in conjunction with one of the other WRITE operations that produce Appian Document outputs (such as [Collection Find to JSON File](#)) this allows for the export and import of larger amounts of data without impacting Process Engine memory usage.

## Parameters

### Output Type

Refer to [Output Type](#) in Common Configuration Parameters.

### Database


Refer to [Database](#) in Common Configuration Parameters.

### Collection

Refer to [Collection](#) in Common Configuration Parameters.

### JSON Source

This parameter allows you to select between passing in JSON values or alternatively reading JSON from an Appian Document.

**JSON Source \*** 

JSON String	▼
Select a Value	
JSON String	
JSON from Appian Document	

### Insert Many JSON Array

*Present only if [JSON Source](#) is “JSON String.”*

The JSON array of MongoDB Documents to be inserted, in the form of: `[{...},{...}]`

### Source JSON File

*Present only if [JSON Source](#) is “JSON from Appian Document.”*

The Appian Document that contains the MongoDB Documents in JSON form to be inserted.

### JSON File Contains a Single Array

*Present only if [JSON Source](#) is “JSON from Appian Document.”*

If Yes (or `true`) then the file will be treated as a single JSON array of MongoDB Documents, e.g. `[{...},{...}]`

If No (or `false`) then the file must have one JSON object per line (delimited with newline), without trailing commas.

## Skip Automatic Date Time Conversion

Present only if [JSON Source](#) is “JSON from Appian Document.”

Refer to [Skip Automatic Date Time Conversion](#) in Common Configuration Parameters.

## Result Value

Field Name	Data Type	Description
database	Text	The database used
collection	Text	The collection used
documentCount	Number (Integer)	The count of new MongoDB Documents inserted into the Collection

## Insert One in Collection

Performs the [Insert One](#) operation, creating a new, singular MongoDB Document instance in the Collection provided.

The screenshot displays the Appian configuration interface for the 'MCSD\_Integration\_insertOne' process. On the left, the configuration panel shows the following settings:

- Connected System:** MCSD MongoDB Connected System
- Operation:** Insert One in Collection
- Output Type:** Dictionary
- Database:** appian
- Collection:** Categories
- Insert One JSON:**

```
1 {
2   category: "Marketing",
3   employeeCount: 62
4 }
```

On the right, the 'Result' tab shows a successful outcome:

- Status:** Success!
- Time:** 18 ms
- Value: Result:**
  - Dictionary
    - database "appian" (Text)
    - collection "Categories" (Text)
    - document Dictionary
      - category "Marketing" (Text)
      - employeeCount 62 (Number (Integer))
      - \_id Dictionary
        - old "5f0c753694849838fddcd7a17" (Text)

## Parameters

### Output Type

Refer to [Output Type](#) in Common Configuration Parameters.

## Database

Refer to [Database](#) in Common Configuration Parameters.

## Collection

Refer to [Collection](#) in Common Configuration Parameters.

## Insert One JSON

The JSON value of the MongoDB Document to be inserted

## Result Value

Field Name	Data Type	Description
database	Text	The database used
collection	Text	The collection used
document	Dictionary OR Text	The MongoDB Document produced by the Insert One operation, either as a Dictionary or JSON string depending on the Output Type selected

## Update Many in Collection

Performs the [Update Many](#) operation, updating all MongoDB Document instances that match the provided Filter JSON.

**MCSD\_Integration\_updateMany** [SAVE CHANGES] [Appian]

☐ Return error if Collection does not exist

**Filter JSON**

```
1 m_query(  
2   m_field(  
3     "category",  
4     m_exists(true)  
5   )  
6 )
```

**m\_query(queryClauses)**  
Begins a query expression  
Returns: Text

**Update Instructions JSON**

```
1 addToSet(  
2   'currentDate': {  
3     lastModified: true,  
4     'updatedOn': {  
5       'type': "timestamp"  
6     }  
7 }  
8 )
```

**TEST REQUEST**

**Result** | Request | Response

**Success!**

**Time**  
10 ms  
Prepare: < 1 ms - Execute: 10 ms (Send/Wait/Receive: 5 ms) - Transform: < 1 ms

**Value: Result**

- Dictionary
  - database "appian" (Text)
  - collection "Categories" (Text)
  - updateResult Dictionary
    - matchedCount 3 (Number (Decimal))
    - modifiedCount 3 (Number (Decimal))
    - upsertedId null (Null)

## Parameters

### Output Type

Refer to [Output Type](#) in Common Configuration Parameters.

### Database

Refer to [Database](#) in Common Configuration Parameters.

### Collection

Refer to [Collection](#) in Common Configuration Parameters.

### Filter JSON

Refer to [Filter JSON](#) in Common Configuration Parameters.

### Update Instructions JSON

This field accepts the JSON to instruct MongoDB how to update the MongoDB Documents matched by the Filter JSON, using [Update Operators](#).

### Skip Automatic Date Time Conversion

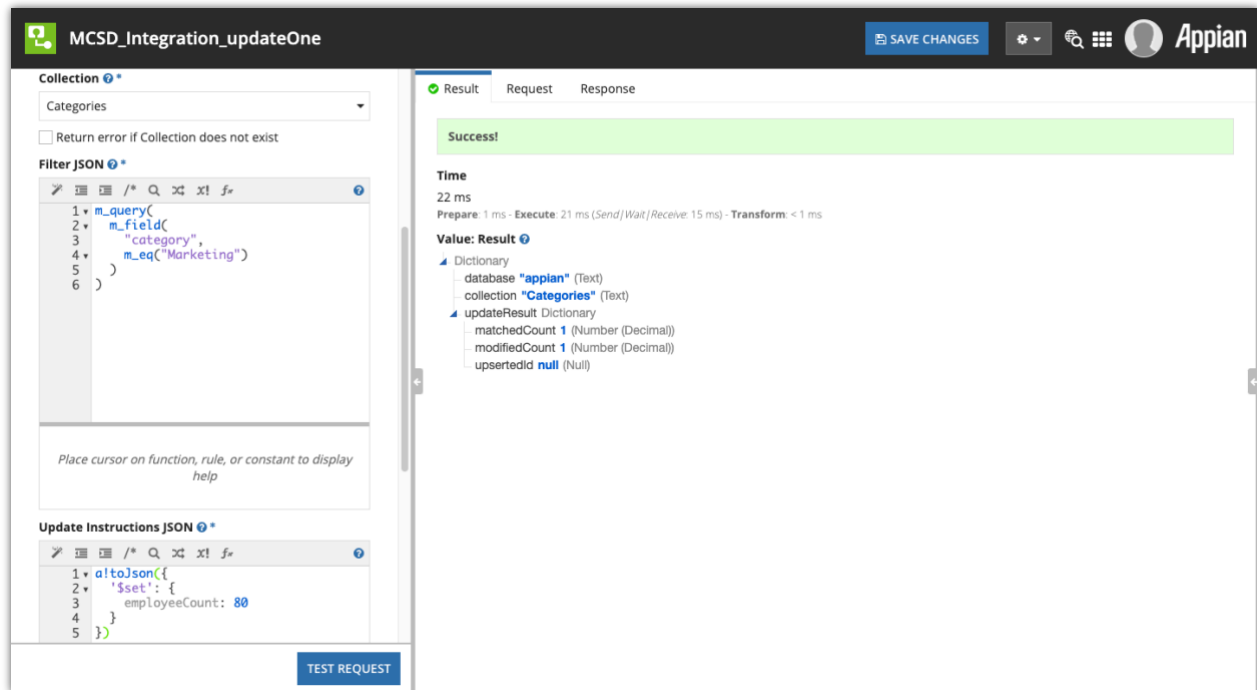
Refer to [Skip Automatic Date Time Conversion](#) in Common Configuration Parameters.

## Result Value

Field Name	Data Type	Description
database	Text	The database used
collection	Text	The collection used
updateResult	Dictionary OR Text	A Dictionary or JSON string (depending on the Output Type selected) that represents the Update Many results as defined in the MongoDB Documentation

# Update One in Collection

Performs the [Update One](#) operation, updating a singular MongoDB Document instance that match the provided Filter JSON.



## Parameters

### Output Type

Refer to [Output Type](#) in Common Configuration Parameters.

### Database

Refer to [Database](#) in Common Configuration Parameters.

### Collection

Refer to [Collection](#) in Common Configuration Parameters.

### Filter JSON

Refer to [Filter JSON](#) in Common Configuration Parameters.

**Note:** This should match a single MongoDB Document, e.g. by filtering on the MongoDB Document's ID (ObjectID).

## Update Instructions JSON

This field accepts the JSON to instruct MongoDB how to update the MongoDB Documents matched by the Filter JSON, using [Update Operators](#).

## Skip Automatic Date Time Conversion

Refer to [Skip Automatic Date Time Conversion](#) in Common Configuration Parameters.

## Result Value

Field Name	Data Type	Description
database	Text	The database used
collection	Text	The collection used
updateResult	Dictionary OR Text	A Dictionary or JSON string (depending on the Output Type selected) that represents the Update One results as defined in the MongoDB Documentation

## Replace One in Collection

Performs the [Replace One](#) operation, completely a singular MongoDB Document instance that match the provided Filter JSON with an entire new MongoDB Document.

The screenshot shows the Appian interface for the 'MCSD\_Integration\_replaceOne' process. The left pane contains two JSON editors: 'Filter JSON' and 'Replacement Mongo Document JSON'. The 'Filter JSON' editor shows a query to find a document by its '\_id'. The 'Replacement Mongo Document JSON' editor shows a new document with 'category: "Marketing"', 'employeeCount: 82', and 'updatedAt: now()'. The right pane displays the 'Result' tab, indicating a successful operation. It shows the execution time (44 ms) and the resulting update statistics: matchedCount 1, modifiedCount 1, and upsertedid null.

```
Filter JSON
1 m_query(
2   m_field(
3     "_id",
4     m_eq(
5       m_objectid("5f0c753694849838fdcc")
6     )
7   )
8 )

Replacement Mongo Document JSON
1 addToJson({
2   category: "Marketing",
3   employeeCount: 82,
4   updatedAt: now()
5 })
```

Result

Success!

Time: 44 ms  
Prepare: <1 ms - Execute: 44 ms (Send/Wait/Receive: 26 ms) - Transform: <1 ms

Value: Result

- Dictionary
  - database "appian" (Text)
  - collection "Categories" (Text)
  - updateResult Dictionary
    - matchedCount 1 (Number (Decimal))
    - modifiedCount 1 (Number (Decimal))
    - upsertedid null (Null)



## Parameters

### Output Type

Refer to [Output Type](#) in Common Configuration Parameters.

### Database

Refer to [Database](#) in Common Configuration Parameters.

### Collection

Refer to [Collection](#) in Common Configuration Parameters.

### Filter JSON

Refer to [Filter JSON](#) in Common Configuration Parameters.

**Note:** This should match a single MongoDB Document, e.g. by filtering on the MongoDB Document's ID (ObjectID).

### Replacement Mongo Document JSON

The JSON value of the new MongoDB Document to replace the one matched.

### Skip Automatic Date Time Conversion

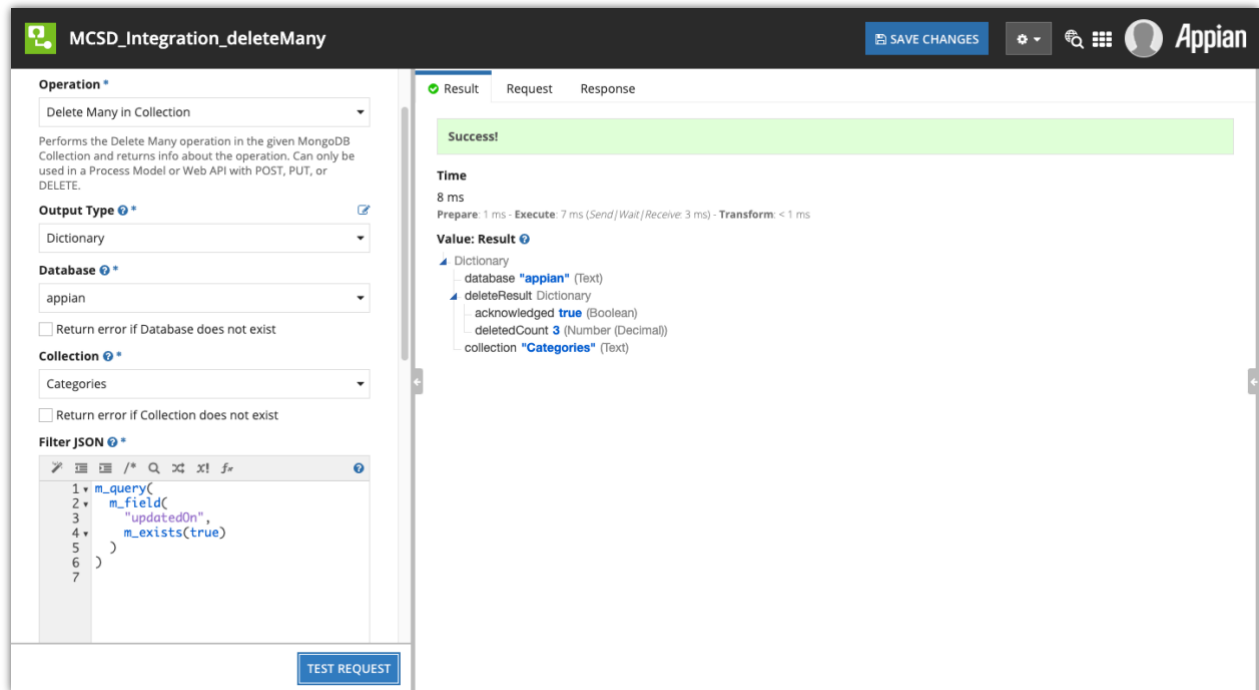
Refer to [Skip Automatic Date Time Conversion](#) in Common Configuration Parameters.

## Result Value

Field Name	Data Type	Description
database	Text	The database used
collection	Text	The collection used
updateResult	Dictionary OR Text	A Dictionary or JSON string (depending on the Output Type selected) that represents the Update One results as defined in the MongoDB Documentation

# Delete Many in Collection

Performs the [Delete Many](#) operation, deleting all MongoDB Document instances that match the provided Filter JSON.



## Parameters

### Output Type

Refer to [Output Type](#) in Common Configuration Parameters.

### Database

Refer to [Database](#) in Common Configuration Parameters.

### Collection

Refer to [Collection](#) in Common Configuration Parameters.

### Filter JSON

Refer to [Filter JSON](#) in Common Configuration Parameters.

**Warning:** It is very important that your Filter JSON matches only the subset of MongoDB Documents to be deleted. There is no “undo” functionality of this operation.

## Collation

Refer to [Collation](#) in Common Configuration Parameters.

## Result Value

Field Name	Data Type	Description
database	Text	The database used
collection	Text	The collection used
deleteResult	Dictionary	Contains the results of the operation with the following 2 keys and values
.acknowledged	Boolean	That the Delete One operation was acknowledged by MongoDB
.deletedCount	Number (Integer)	The number of MongoDB Documents deleted by this operation.

## Delete One in Collection

Performs the [Delete One](#) operation, updating a singular MongoDB Document instance that match the provided Filter JSON.

The screenshot displays the Appian interface for the 'MCSD\_Integration\_deleteOne' operation. The configuration panel on the left shows the operation set to 'Delete One in Collection', with the output type as 'Dictionary'. The database is 'appian' and the collection is 'Categories'. The filter JSON is configured to delete a document with a specific object ID. The right panel shows the 'Result' tab, indicating a 'Success!' status. The response details show a time of 7 ms and a value result dictionary containing 'database: appian', 'deleteResult' (a dictionary with 'acknowledged: true' and 'deletedCount: 1'), and 'collection: Categories'.

```
1 • m_query(  
2   m_field(  
3     "_id",  
4     m_eq(  
5       m_objectid("5f0dc87db0b73525bb95"  
6     )  
7   )  
8 )
```

TEST REQUEST

Result Request Response

Success!

Time  
7 ms  
Prepare: < 1 ms - Execute: 7 ms (Send/Wait/Receive: 1 ms) - Transform: < 1 ms

Value: Result

Dictionary  
- database "appian" (Text)  
- deleteResult Dictionary  
 - acknowledged true (Boolean)  
 - deletedCount 1 (Number (Decimal))  
- collection "Categories" (Text)

## Parameters

### Output Type

Refer to [Output Type](#) in Common Configuration Parameters.

### Database

Refer to [Database](#) in Common Configuration Parameters.

### Collection

Refer to [Collection](#) in Common Configuration Parameters.

### Filter JSON

Refer to [Filter JSON](#) in Common Configuration Parameters.

**Warning:** It is very important that your Filter JSON matches only the **single** MongoDB Document to be deleted. There is no “undo” functionality of this operation.

### Collation

Refer to [Collation](#) in Common Configuration Parameters.

## Result Value

Field Name	Data Type	Description
<code>database</code>	Text	The database used
<code>collection</code>	Text	The collection used
<code>deleteResult</code>	Dictionary	Contains the results of the operation with the following 2 keys and values
<code>.acknowledged</code>	Boolean	That the Delete One operation was acknowledged by MongoDB
<code>.deletedCount</code>	Number (Integer)	The number of MongoDB Documents deleted by this operation (should always be 1 for Delete One).

# Drop Collection

Performs the [Drop](#) operation on a Collection, deleting the Collection and any MongoDB Documents found within.

**Warning:** This operation can be very destructive if not used with great caution. There is no “undo” functionality of this operation.

**Connected System \***  
MCSDB MongoDB Connected System

**Operation \***  
Drop Collection

**Database**  
r!databaseName  
☒ Return error if Database does not exist

**Collection**  
r!collectionName  
☐ Return error if Collection does not exist

Rule Input Name	Expression	Value
databaseName (Text)	1	applan
collectionName (Text)	1	Categories

Set as default test values

**Result** Request Response

Success!

Time  
22 ms  
Prepare: < 1 ms - Execute: 21 ms (Send/Wait/Receive: 17 ms) - Transform: 1 ms

Value: Result

Dictionary

- database "applan" (Text)
- collection "Categories" (Text)
- collectionDropped true (Boolean)

## Parameters

### Database

Refer to [Database](#) in Common Configuration Parameters.

### Collection

Refer to [Collection](#) in Common Configuration Parameters.

## Result Value

Field Name	Data Type	Description
database	Text	The database used
collection	Text	The collection used
collectionDropped	Boolean	Whether the Collection was successfully dropped

# Common Configuration Parameters

This section describes the collection of Parameters that are shared among multiple Integration Operations.

## Output Type

This parameter allows you to select how the Integration Operation returns the data, either as Appian Dictionaries or as a List of Text containing the JSON representations of the MongoDB Documents. See [Special Considerations](#) for how ObjectIds are transformed when being output as Dictionaries.

When selecting the List of JSON Strings, the JSON output comes directly from the MongoDB Java Driver, which represents the data in its “purest” form.

**Output Type** ? \*

Dictionary

Select a Value

Dictionary

List of JSON Strings

## Database

This parameter tells the Integration Operation which MongoDB Database to perform the operation on. The drop-down is automatically populated with the available Databases in your Connected System, using the same method as [List Databases](#) above.

Below the drop-down is a checkbox which if checked will ensure that the Database exists before performing the Operation. MongoDB is very forgiving and will allow many API methods to be performed on non-existent Databases, returning null or an empty set. Checking this box will ensure that the database exists, to help avoid entering the wrong database name.

**Database** ? \*

Select a Value

☐ Return error if Database does not exist

## Collection

This parameter tells the Integration Operation which MongoDB Collection to perform the operation on. The drop-down is automatically populated with the available Collections in the given Database in your Connected System, using the same method as [List Collections](#) above.

Below the drop-down is a checkbox which if checked will ensure that the Collection exists before performing the Operation. MongoDB is very forgiving and will allow many API methods to be performed on non-existent Collections, returning null or an empty set. Checking this box will ensure that the Collection exists, to help avoid entering the wrong Collection name.

Collection ? \*

Select a Value

☐ Return error if Collection does not exist

**Note:** For WRITE operations (e.g. [Insert One](#)), if the checkbox is unchecked and the Collection does not exist, MongoDB will create a new Collection with the given name to perform the write into. This can be an unintended action, but it can also be very useful when performing certain business functions, such as reading back the results of an [Aggregate](#) and writing them to a new Collection in one nested operation.

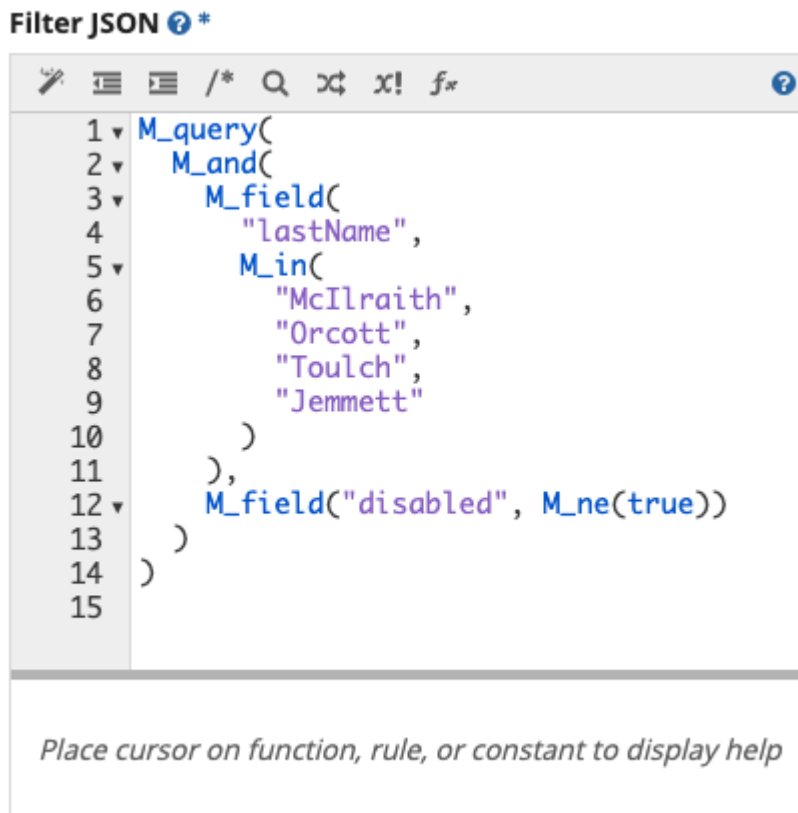
## Filter JSON

This parameter represents a [MongoDB Query Document](#) in JSON form. This is the query language that MongoDB uses, similar to how RDBMS platforms use SQL. When present, the value provided here will be used to match MongoDB Documents in the given Collection.

For READ operations such as Find, this will determine which MongoDB Documents are returned.

For WRITE operations such as Update, this will determine which MongoDB Documents are modified, so it is critical your Filter JSON matches only those to be modified. It is therefore a good idea to test your Filter JSON values for WRITE operations before performing them (e.g. by performing a Find first and validating that only the documents to be updated are returned).

This JSON string value can be hard-coded, constructed via Expressions or Expression Rules, or constructed using the [JSON Query Expression Functions](#) as shown:



## Output JSON As a Single Array

This will join the results as a JSON array such as: `[{...},{...}]`, allowing easier import using [Insert Many in Collection](#) or other MongoDB tools. Selecting No (or `false`) will write one MongoDB Document per line.

## Save to Folder

The Appian Folder where the new Document will be created. This is a standard Appian folder picker.

## Filename

The complete name of the output file, including any extension you would like (most commonly `.json`)



## Skip Automatic Date Time Conversion

This plugin will attempt to detect Dates and Times and convert them to MongoDB timestamps. Selecting Yes (or `true`) here will skip this and instead insert these values as Strings.

## Character Set

The character set that the text file should use. Valid values are:

Value	Description
ISO-8859-1	ISO Latin Alphabet No. 1, a.k.a. ISO-LATIN-1
UTF-8	Eight-bit UCS Transformation Format
UTF-16LE	Sixteen-bit UCS Transformation Format, little-endian byte order
UTF-16	Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark
UTF-16BE	Sixteen-bit UCS Transformation Format, big-endian byte order
US-ASCII	Seven-bit ASCII, a.k.a. ISO646-US, a.k.a. the Basic Latin block of the Unicode character set

### Character Set ? \*

Select a Value

Select a Value

ISO-8859-1

UTF-8

UTF-16LE

UTF-16

UTF-16BE

US-ASCII

## Collation

[Collation](#) allows users to specify language-specific rules for string comparison, such as rules for letter case and accent marks. This parameter section allows you to individually configure a

[Collation Document](#), or you can define one by building an Appian Dictionary (and not a JSON string) in the form of:

```
{
  COLLATION_LOCALE: "text",
  COLLATION_CASE_LEVEL: true,
  COLLATION_CASE_FIRST: "text",
  COLLATION_STRENGTH: 100,
  COLLATION_NUMERIC_ORDERING: true,
  COLLATION_ALTERNATE: "text",
  COLLATION_MAX_VARIABLE: "text",
  COLLATION_BACKWARDS: true
}
```

#### Collation ?

- ☒ Specify values for each input  
☐ Define all values with a single expression

Name	Type	Value
Locale	Text	<input type="text"/>
Case Level	Boolean	<input type="radio"/> Yes <input checked="" type="radio"/> No
Case First	Text	<input type="text"/>
Strength	Number (Integer)	<input type="text"/>
Numeric Ordering	Boolean	<input type="radio"/> Yes <input checked="" type="radio"/> No
Alternate	Text	<input type="text"/>
Max Variable	Text	<input type="text"/>
Backwards	Boolean	<input type="radio"/> Yes <input checked="" type="radio"/> No

## Read Preference

[Read Preference](#) describes how MongoDB clients route read operations to the members of a replica set.

## Read Concern

[Read Concern](#) allows you to control the consistency and isolation properties of the data read from replica sets and replica set shards.

# JSON Query Expression Functions

As many Integration Operation parameters take as input JSON expressions, such as those for filtering MongoDB Documents in [Collection Find](#), we have included a set of Expression Functions to help create these JSON Query expressions without having to construct the strings yourself.

The screenshot displays the Appian IDE interface. On the left, a rule named 'rule!MCSD\_Integration\_collectionFind' is defined. The rule's filter is a JSON query expression: `filterJson: M_query(M_and(M_field("lastName", M_in("McIlraith", "Orcott", "Toulch", "Jemmett")), M_field("disabled", M_exists(false))))`. The limit is set to 10. On the right, the 'Test Output' panel shows the result of the query. The output is a Dictionary with the following structure: `{ success: true, result: { database: 'appian', collection: 'Customers', documents: [ { _id: { old: '5efa0b04fc13ae730e000064', createdOn: '2/16/2019 3:58 PM EST', createdBy: 'mongodbTestUser', firstName: 'Jenny', lastName: 'McIlraith' } } ] }`.

In addition to generating the special JSON structures MongoDB expects, these functions make creating MongoDB's non-standard JSON alterations much easier in Appian. The functions also handle converting Appian primitive types to their necessary JSON representations. For example, this expression:

```
M_query(  
  M_field(  
    "createdOn",  
    M_eq(now())  
  )  
)
```

Would produce this JSON:

```
{ "createdOn": { "$eq": ISODate("2020-07-01T20:32:20.900Z") } }
```

Note the MongoDB-specific JSON `ISODate()` function call. Also note the field names that begin with `$`. It is not easy (and, in some cases such as this one, not possible) to generate JSON in this manner by constructing Dictionaries and using Appian's built-in `addToJson()` function.

More MongoDB query functions will be added in later versions of this plugin.

# Top-Level Functions

## M\_query()

This function begins a MongoDB query. It should be the top-level function call whose output is sent to one of the JSON filter parameters. Essentially this wraps the contents provided in braces { ... } to ensure a complete query JSON string.

**queryClauses** (List of Text String): The list of expressions (often created with [M\\_Field](#)) you wish to evaluate

For example, this expression:

```
M_query(  
  M_field("createdOn", M_eq(fn!datetime(2019,4,26,10,28,57,0)))  
)
```

Would produce this JSON:

```
{ "createdOn": { "$eq": ISODate("2019-04-26T10:28:57.000Z") } }
```

## M\_field()

This function begins a query expression on a field, in the form of fieldName: ... where the passed in queryClauses are joined to complete the expression. To be used within M\_Query or one of the other

**field** (Text): The name of the field in the MongoDB Document you wish to filter on

**queryClauses** (List of Text String): The list of expressions (often created with other m\_\* functions) you wish to evaluate

For example, this expression:

```
M_field("createdOn", M_eq(fn!datetime(2019,4,26,10,28,57,0)))
```

Would produce this portion of JSON:

```
"createdOn": { "$eq": ISODate("2019-04-26T10:28:57.000Z") }
```

# Comparison Query Operators

These functions correspond directly to the [Comparison Query Operators](#) provided by the MongoDB Query language.

These functions handle converting Appian primitive types to their necessary JSON representations. For example, this expression:

```
M_query(  
  M_field(  
    "createdOn",  
    M_eq(now())  
  )  
)
```

Would produce this JSON:

```
{ "createdOn": { "$eq": ISODate("2020-07-01T20:32:20.900Z") } }
```

Note the MongoDB-specific `ISODate()` function call. It is not possible to generate JSON in this manner using Appian's built-in `!toJson()` function.

## M\_eq()

Implements the `$eq` operator. Specifies equality condition. The `$eq` operator matches documents where the value of a field equals the specified value.

**value** (Any Type): The value to evaluate against.

## M\_gt()

Implements the `$gt` operator. Selects those documents where the value of the field is greater than (i.e. `>`) the specified value.

**value** (Any Type): The value to evaluate against.

## M\_gte()

Implements the `$gte` operator. Selects the documents where the value of the field is greater than or equal to (i.e. `>=`) a specified value (e.g. value.)

**value** (Any Type): The value to evaluate against.

## M\_in()

Implements the `$in` operator. Selects the documents where the value of a field equals any value in the specified array.

**array** (List of Variant): The array of values to evaluate against.

## M\_lt()

Implements the `$lt` operator. Selects the documents where the value of the field is less than (i.e. `<`) the specified value.

**value** (Any Type): The value to evaluate against.

## M\_lte()

Implements the `$lte` operator. Selects the documents where the value of the field is less than or equal to (i.e. `<=`) the specified value.

**value** (Any Type): The value to evaluate against.

## M\_ne()

Implements the `$ne` operator. Selects the documents where the value of the field is not equal to the specified value. This includes documents that do not contain the field.

**value** (Any Type): The value to evaluate against.

## M\_nin()

Implements the `$nin` operator. Selects the documents where the field value is not in the specified array or the field does not exist.

**array** (List of Variant): The array of values to evaluate against.

# Logical Query Operators

These functions correspond directly to the [Logical Query Operators](#) provided by the MongoDB Query language.

For example, this expression:

```
M_query(  
  M_and(  
    M_field(  
      "lastName",  
      M_in(  
        "McIlraith",  
        "Orcott",  
        "Toulch",  
        "Jemmett"  
      )  
    ),  
    M_field("disabled", M_ne(true))  
  )  
)
```

Would produce this JSON:

```
{ "$and": [ { "lastName": { "$in": [ "McIlraith", "Orcott", "Toulch",  
  "Jemmett" ] } }, { "disabled": { "$ne": true } } ] }
```

## M\_and()

Implements the [\\$and](#) operator. Performs a logical AND operation on an array of one or more expressions (e.g. expression1, expression2, etc.) and selects the documents that satisfy all the expressions in the array. The [\\$and](#) operator uses short-circuit evaluation. If the first expression (e.g. expression1) evaluates to false, MongoDB will not evaluate the remaining expressions.

**queryExpressions** (List of Text String): The list of expressions (often created with other M\_\* functions) you wish to evaluate against.

## M\_nor()

Implements the [\\$nor](#) operator. Performs a logical NOR operation on an array of one or more query expressions and selects the documents that fail all the query expressions in the array.



**queryExpressions** (List of Text String): The list of expressions (often created with other M\_\* functions) you wish to evaluate against.

## M\_not()

Implements the [\\$not](#) operator. Performs a logical NOT operation on the specified operator-expression and selects the documents that do not match the operator-expression. This includes documents that do not contain the field.

**queryExpression** (Text): The expression (often created with other M\_\* functions) you wish to evaluate against.

## M\_or()

Implements the [\\$or](#) operator. Performs a logical OR operation on an array of two or more expressions and selects the documents that satisfy at least one of the expressions.

**queryExpressions** (List of Text String): The list of expressions (often created with other M\_\* functions) you wish to evaluate against.

# Element Query Operators

These functions correspond directly to the [Element Query Operators](#) provided by the MongoDB Query language.

## M\_exists()

Implements the [\\$exists](#) operator. When value is true, [\\$exists](#) matches the documents that contain the field, including documents where the field value is null. If value is false, the query returns only the documents that do not contain the field.

**value** (Boolean): Whether it should exist or not.

## M\_type()

Implements the [\\$type](#) operator. Selects documents where the value of the field is an instance of the specified BSON type(s). Querying by data type is useful when dealing with highly unstructured data where data types are not predictable.

**types** (List of Variant): Either the BSON type numbers (integer) or aliases (string).

# Evaluation Query Operators

These functions correspond directly to the [Evaluation Query Operators](#) provided by the MongoDB Query language.

## `M_expr()`

Implements the [\\$expr](#) operator. Allows the use of aggregation expressions within the query language.

**queryExpression** (Text): The expression (often created with other `M_*` functions) you wish to evaluate against.

## `M_jsonSchema()`

Implements the [\\$jsonSchema](#) operator. Matches documents that satisfy the specified JSON Schema.

**jsonSchema** (Text): The JSON Schema object, formatted according to [draft 4 of the JSON Schema standard](#). Must be enclosed in brackets ({}).

## `M_mod()`

Implements the [\\$mod](#) operator. Select documents where the value of a field divided by a divisor has the specified remainder (i.e. perform a modulo operation to select documents).

**divisor** (Number (Integer)): The divisor value.

**remainder** (Number (Integer)): The remainder value.

## `M_regex()`

Implements the [\\$regex](#) operator. Provides regular expression capabilities for pattern matching strings in queries. MongoDB uses Perl compatible regular expressions (i.e. "PCRE" ) version 8.42 with UTF-8 support.

**regex** (Text): The regular expression (without enclosing slashes), e.g. `"^foo.*bar$"`

**options** (Text): The regular expression options modifiers (`"i"`, `"m"`, `"s"`, and/or `"x"`), e.g. `"im"` for 'ignore case' and 'multiline' searches

This example would match all MongoDB Documents where the last name begins with “St”:

```
M_query(  
  M_field(  
    "lastName",  
    M_regex("^St", "i")  
  )  
)
```

Which would produce this JSON:

```
{ "lastName": { "$regex": /^St/i } }
```

## M\_text()

Implements the [\\$text](#) operator. Performs a text search on the content of the fields indexed with a text index.

**search** (Text): A string of terms that MongoDB parses and uses to query the text index. MongoDB performs a logical OR search of the terms unless specified as a phrase.

**language** (Text): Optional (use null to omit). The language that determines the list of stop words for the search and the rules for the stemmer and tokenizer. If not specified, the search uses the default language of the index.

**caseSensitive** (Boolean): Optional (use null to omit). A boolean flag to enable or disable case sensitive search. Defaults to false; i.e. the search defers to the case insensitivity of the text index.

**diacriticSensitive** (Boolean): Optional (use null to omit). A boolean flag to enable or disable diacritic sensitive search against version 3 text indexes. Defaults to false; i.e. the search defers to the diacritic insensitivity of the text index.

## M\_where()

Implements the [\\$where](#) operator. Use the `$where` operator to pass either a string containing a JavaScript expression or a full JavaScript function to the query system. The `$where` provides greater flexibility, but requires that the database processes the JavaScript expression or function for each document in the collection. Reference the document in the JavaScript expression or function using either `this` or `obj`.

Please see [full documentation](#) for caveats and performance topics.

**javascript** (Text): A JavaScript expression or a full JavaScript function.

This example would match all MongoDB Documents where the last name equals “Gudgen”:

```
M_query(  
  M_where("function() { return (hex_md5(this.lastName) ==  
  '9af26c4c8b156852e86d49566d96a0d1' ) }")  
)
```

Which would produce this JSON:

```
{ "$where": "function() { return (hex_md5(this.lastName) ==  
\'9af26c4c8b156852e86d49566d96a0d1\') }" }
```

## Query Operator Examples

### Date and Time Examples

#### Query for Dates In a Range

This expression will produce a query for finding MongoDB Documents with `createdOn` in the month of December 2019:

```
M_query(  
  M_field(  
    "createdOn",  
    M_gte(date(2019, 12, 1)),  
    M_lt(date(2020, 1, 1))  
  )  
)
```

Which produces this JSON:

```
{  
  "createdOn": {  
    "$gte": ISODate("2019-12-01T00:00:00.000Z"),  
    "$lt": ISODate("2020-01-01T00:00:00.000Z")  
  }  
}
```

## Querying by Date Without Time

As mentioned earlier, MongoDB stores all dates in UTC and does not have a date without time. Similar to above, in order to find all MongoDB Documents with `createdOn` on a single day, use `$gte` of that day:

```
M_query(  
  M_field(  
    "createdOn",  
    M_gte(date(2019, 12, 1)),  
    M_lt(date(2020, 1, 1))  
  )  
)
```

Which produces this JSON:

```
{  
  "createdOn": {  
    "$gte": ISODate("2019-12-01T00:00:00.000Z"),  
    "$lt": ISODate("2020-01-01T00:00:00.000Z")  
  }  
}
```

More examples will be added in newer versions of this plugin.

# Changelog

Date	Version	Description
2020-07-14	1.0	Initial version

## Future Enhancements

As of this writing, this list comprises the next known enhancements to this plugin. If you do not see an enhancement here that is needed, please comment on the [Community App Market](#) entry.

- An example Appian Application that uses the MongoDB Connected System in Expression-Backed records, plus other examples such as using Aggregates for producing charts or other reports.
- More Query Operator Examples
- Support for Write Concern in all supported operations
- Add support for UpdateOptions in Update One and Update Many
- Add support of Aggregate Pipelines in Update operations
- More JSON operator Expression Functions:
  - All remaining MongoDB Query Operators
  - Update Operators
  - Aggregate Pipelines