MongoDB Tutorial



MongoDB tutorial provides basic and advanced concepts of SQL. Our MongoDB tutorial is designed for beginners and professionals.

MongoDB is a No SQL database. It is an open-source, cross-platform, document-oriented database written in C++.

Our MongoDB tutorial includes all topics of MongoDB database such as insert documents, update documents, delete documents, query documents, projection, sort() and limit() methods, create a collection, drop collection, etc. There are also given MongoDB interview questions to help you better understand the MongoDB database.

What is MongoDB

[MongoDB](https://www.javatpoint.com/mongodb-tutorial) is an open-source document database that provides high performance, high availability, and automatic scaling.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

In simple words, you can say that - Mongo DB is a document-oriented database. It is an open source product, developed and supported by a company named 10gen.

MongoDB is available under General Public license for free, and it is also available under Commercial license from the manufacturer.

The manufacturing company 10gen has defined MongoDB as:

"MongoDB is a scalable, open source, high performance, document-oriented database." - 10gen

MongoDB was designed to work with commodity servers. Now it is used by the company of all sizes, across all industry.

History of MongoDB

The initial development of MongoDB began in 2007 when the company was building a platform as a service similar to window azure.

Window azure is a cloud computing platform and infrastructure, created by Microsoft, to build, deploy and manage applications and service through a global network.

MongoDB was developed by a NewYork based organization named 10gen which is now known as MongoDB Inc. It was initially developed as a PAAS (Platform as a Service). Later in 2009, it is introduced in the market as an open source database server that was maintained and supported by MongoDB Inc.

The first ready production of MongoDB has been considered from version 1.4 which was released in March 2010.

MongoDB2.4.9 was the latest and stable version which was released on January 10, 2014.

Purpose of Building MongoDB

It may be a very genuine question that - "what was the need of MongoDB although there were many databases in action?"

**There is a simple answer:**

All the modern applications require big data, fast features development, flexible deployment, and the older database systems not competent enough, so the MongoDB was needed.

**The primary purpose of building MongoDB is:**

* Scalability
* Performance
* High Availability
* Scaling from single server deployments to large, complex multi-site architectures.
* Key points of MongoDB
* Develop Faster
* Deploy Easier
* Scale Bigger

First of all, we should know what is document oriented database?

Example of Document-Oriented Database

MongoDB is a document-oriented database. It is a key feature of MongoDB. It offers a document-oriented storage. It is very simple you can program it easily.

MongoDB stores data as documents, so it is known as document-oriented database.

1. FirstName = "John",
2. Address = "Detroit",
3. Spouse = [{**Name**: "Angela"}].
4. FirstName ="John",
5. Address = "Wick"

**There are two different documents (separated by ".").**

Storing data in this manner is called as document-oriented database.

Mongo DB falls into a class of databases that calls Document Oriented Databases. There is also a broad category of database known as [No SQL Databases](https://www.javatpoint.com/nosql-databases).

Features of MongoDB

These are some important features of MongoDB:

**1. Support ad hoc queries**

In MongoDB, you can search by field, range query and it also supports regular expression searches.

**2. Indexing**

You can index any field in a document.

**3. Replication**

MongoDB supports Master Slave replication.

A master can perform Reads and Writes and a Slave copies data from the master and can only be used for reads or back up (not writes)

**4. Duplication of data**

MongoDB can run over multiple servers. The data is duplicated to keep the system up and also keep its running condition in case of hardware failure.

**5. Load balancing**

It has an automatic load balancing configuration because of data placed in shards.

**6. Supports map reduce and aggregation tools**.

**7. Uses**[**JavaScript**](https://www.javatpoint.com/javascript-tutorial)**instead of Procedures**.

**8. It is a schema-less database written in**[**C++**](https://www.javatpoint.com/cpp-tutorial).

**9. Provides high performance**.

**10. Stores files of any size easily without complicating your stack**.

**11. Easy to administer in the case of failures**.

**12. It also supports:**

* JSON data model with dynamic schemas
* Auto-sharding for horizontal scalability
* Built in replication for high availability
* Now a day many companies using MongoDB to create new types of applications, improve performance and availability.

Prerequisite

Before learning MongoDB, you must have the basic knowledge of SQL and OOPs.

Audience

Our MongoDB tutorial is designed to help beginners and professionals.

Problem

We assure that you will not find any problem in this MongoDB tutorial. But if there is any mistake, please post the problem in contact form.

NoSQL Databases

We know that MongoDB is a NoSQL Database, so it is very necessary to know about NoSQL Database to understand MongoDB throughly.

What is NoSQL Database

Databases can be divided in 3 types:

1. RDBMS (Relational Database Management System)
2. OLAP (Online Analytical Processing)
3. NoSQL (recently developed database)

NoSQL Database

NoSQL Database is used to refer a non-SQL or non relational database.

It provides a mechanism for storage and retrieval of data other than tabular relations model used in relational databases. NoSQL database doesn't use tables for storing data. It is generally used to store big data and real-time web applications.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

History behind the creation of NoSQL Databases

In the early 1970, Flat File Systems are used. Data were stored in flat files and the biggest problems with flat files are each company implement their own flat files and there are no standards. It is very difficult to store data in the files, retrieve data from files because there is no standard way to store data.

Then the relational database was created by E.F. Codd and these databases answered the question of having no standard way to store data. But later relational database also get a problem that it could not handle big data, due to this problem there was a need of database which can handle every types of problems then NoSQL database was developed.

Advantages of NoSQL

* It supports query language.
* It provides fast performance.
* It provides horizontal scalability.

MongoDB advantages over RDBMS

In recent days, MongoDB is a new and popularly used database. It is a document based, non relational database provider.

Although it is 100 times faster than the traditional database but it is early to say that it will broadly replace the traditional RDBMS. But it may be very useful in term to gain performance and scalability.

A Relational database has a typical schema design that shows number of tables and the relationship between these tables, while in MongoDB there is no concept of relationship.

MongoDB Advantages

* **MongoDB is schema less**. It is a document database in which one collection holds different documents.
* There may be **difference between number of fields, content and size of the document** from one to other.
* **Structure of a single object is clear** in MongoDB.
* There are **no complex joins** in MongoDB.
* MongoDB provides the **facility of deep query** because it supports a powerful dynamic query on documents.
* It is very **easy to scale**.
* It **uses internal memory for storing working sets** and this is the reason of its fast access.

Distinctive features of MongoDB

* Easy to use
* Light Weight
* Extremely faster than RDBMS

Where MongoDB should be used

* Big and complex data
* Mobile and social infrastructure
* Content management and delivery
* User data management
* Data hub

Performance analysis of MongoDB and RDBMS

* In relational database (RDBMS) tables are using as storing elements, while in MongoDB collection is used.
* In the RDBMS, we have multiple schema and in each schema we create tables to store data while, MongoDB is a document oriented database in which data is written in BSON format which is a JSON like format.
* MongoDB is almost 100 times faster than traditional database systems.

MongoDB Datatypes

Following is a list of usable data types in MongoDB.

|  |  |
| --- | --- |
| **Data Types** | **Description** |
| String | String is the most commonly used datatype. It is used to store data. A string must be UTF 8 valid in mongodb. |
| Integer | Integer is used to store the numeric value. It can be 32 bit or 64 bit depending on the server you are using. |
| Boolean | This datatype is used to store boolean values. It just shows YES/NO values. |
| Double | Double datatype stores floating point values. |
| Min/Max Keys | This datatype compare a value against the lowest and highest bson elements. |
| Arrays | This datatype is used to store a list or multiple values into a single key. |
| Object | Object datatype is used for embedded documents. |
| Null | It is used to store null values. |
| Symbol | It is generally used for languages that use a specific type. |
| Date | This datatype stores the current date or time in unix time format. It makes you possible to specify your own date time by creating object of date and pass the value of date, month, year into it. |

# How to install MongoDB on Windows

Firstly you will have to download the latest release of MongoDB:

## How to download MongoDB

You can download an appropriate version of MongoDB which your system supports, from the link **"http://www.mongodb.org/downloads"** to install the MongoDB on Windows. You should choose correct version of MongoDB acording to your computer's Window. If you are not sure what Window version are you using, open your command prompt and execute this command:

1. C:\ wmic os get osarchitecture

OSArchitecture

64 bit

C:\ >

#### **Note: MongoDB does not support Window XP.**

## MongoDB for Windows Server 2008 R2 edition

This version of MongoDB runs only on Window Server 2008 R2, Window7 64 bit, and the newer version of windows. You can't operate it on older version of windows.

## MongoDb for 64 bit Windows

This version of MongoDB runs only on newer version of Windows contains 64 bit operating system.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

for example: Window Server 2008 R2, Window 7 64 bit etc.

## MongoDb for 32 bit Windows

This version of MongoDB runs on only 32 bit windows. 32 bit version of MongoDB is generally used in testing and development purposes because it supports databases smaller than 2 GB.

## How to install the downloaded file

In Window explorer, locate the downloaded MongoDB msi file, double click on that file and follow the instructions appears on the screen. These instructions will guide you to complete the installation process.

#### **Note: If you want to move the MongoDB folder from default position to another position, it is necessary to issue the move command as an administrator. let us take an example to move the folder to C : \mongodb:**

1. **Select** Start Menu > All Programs > Accessories

You can install MongoDB in any folder because it is self contained and does not have any other system dependency.

## How to set up the MongoDB environment

A data directory is required in MongoDB to store all the information. Its by default data directory path is \data\db. you can create this folder by command prompt.

1. md\data\db

**For example:**

If you want to start MongoDB, run mongod.exe

You can do it from command prompt.

1. C:\Program Files\MongoDB\bin\mongod.exe

This will start the mongoDB database process. If you get a message "waiting for connection" in the console output, it indicates that the mongodb.exe process is running successfully.

**For example:**

When you connect to the MongoDB through the mongo.exe shell, you should follow these steps:

1. Open another command prompt.
2. At the time of connecting, specify the data directory if necessary.

#### **Note: If you use the default data directory while MongoDB installation, there is no need to specify the data directory.**

**For example:**

1. C:\mongodb\bin\mongo.exe

If you use the different data directory while MongoDB installation, specify the directory when connecting.

**For example:**

1. C:\mongodb\bin\mongod.exe-- dbpath d:\test\mongodb\data

If you have spaces in your path, enclose the entire path in double space.

**For example:**

1. C:\mongodb\bin\mongod.exe-- dbpath  "d: \ test\mongodb\data"

## How to configure directory and files

First to create a configuration file and a directory path for MongoDB log output after that create a specific directory for MongoDB log files.

Data Modeling in MongoDB

In MongoDB, data has a flexible schema. It is totally different from SQL database where you had to determine and declare a table's schema before inserting data. MongoDB collections do not enforce document structure.

The main challenge in data modeling is balancing the need of the application, the performance characteristics of the database engine, and the data retrieval patterns.

Consider the following things while designing the schema in MongoDB

* Always design schema according to user requirements.
* Do join on write operations not on read operations.
* Objects which you want to use together, should be combined into one document. Otherwise they should be separated (make sure that there should not be need of joins).
* Optimize your schema for more frequent use cases.
* Do complex aggregation in the schema.
* You should duplicate the data but in a limit, because disc space is cheaper than compute time.

**For example:**

let us take an example of a client who needs a database design for his website. His website has the following requirements:

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

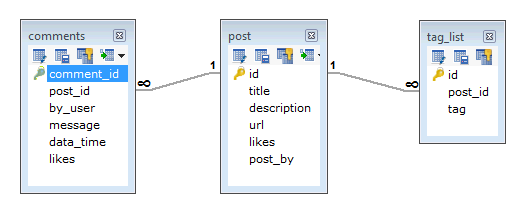
Every post is distinct (contains unique title, description and url).

Every post can have one or more tags.

Every post has the name of its publisher and total number of likes.

Each post can have zero or more comments and the comments must contain user name, message, data-time and likes.

For the above requirement, a minimum of three tables are required in RDBMS.



But in MongoDB, schema design will have one collection post and has the following structure:

{

\_id: POST\_ID

title: TITLE\_OF\_POST,

description: POST\_DESCRIPTION,

by: POST\_BY,

url: URL\_OF\_POST,

tags: [TAG1, TAG2, TAG3],

likes: TOTAL\_LIKES,

comments: [

{

user: 'COMMENT\_BY',

message: TEXT,

datecreated: DATE\_TIME,

like: LIKES

},

{

user: 'COMMENT\_BY',

message: TEST,

dateCreated: DATE\_TIME,

like: LIKES

}}}

# MongoDB Create Database

**Use Database method:**

There is no create database command in MongoDB. Actually, MongoDB do not provide any command to create database.

It may be look like a weird concept, if you are from traditional SQL background where you need to create a database, table and insert values in the table manually.

Here, in MongoDB you don't need to create a database manually because MongoDB will create it automatically when you save the value into the defined collection at first time.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

You also don't need to mention what you want to create, it will be automatically created at the time you save the value into the defined collection.

#### **Here one thing is very remarkable that you can create collection manually by "db.createCollection()" but not the database.**

## How and when to create database

If there is no existing database, the following command is used to create a new database.

**Syntax:**

1. use DATABASE\_NAME

If the database already exists, it will return the existing database.

Let' take an example to demonstrate how a database is created in [MongoDB](https://www.javatpoint.com/mongodb-tutorial). In the following example, we are going to create a database "javatpointdb".

**See this example**

1. >use javatpointdb

Swithched to db javatpointdb

To **check the currently selected database**, use the command db:

1. >db

javatpointdb

To **check the database list**, use the command show dbs:

1. >show dbs

local 0.078GB

Here, your created database "javatpointdb" is not present in the list, **insert at least one document** into it to display database:

1. >db.movie.**insert**({"name":"javatpoint"})

WriteResult({ "nInserted": 1})

1. >show dbs

javatpointdb 0.078GB

local 0.078GB

# MongoDB Drop Database

The dropDatabase command is used to drop a database. It also deletes the associated data files. It operates on the current database.

**Syntax:**

1. db.dropDatabase()

This syntax will delete the selected database. In the case you have not selected any database, it will delete default "test" database.

To **check the database list**, use the command show dbs:

1. >show dbs

javatpointdb 0.078GB

local 0.078GB

If you want to **delete the database "javatpointdb"**, use the dropDatabase() command as follows:

1. >use javatpointdb

switched to the db javatpointdb

1. >db.dropDatabase()

{ "dropped": "javatpointdb", "ok": 1}

Now check the list of databases:

1. >show dbs

local 0.078GB

MongoDB Create Collection

In MongoDB, db.createCollection(name, options) is used to create collection. But usually you don?t need to create collection. MongoDB creates collection automatically when you insert some documents. It will be explained later. First see how to create collection:

**Syntax:**

1. db.createCollection(**name**, options)

Here,

**Name:** is a string type, specifies the name of the collection to be created.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

**Options:** is a document type, specifies the memory size and indexing of the collection. It is an optional parameter.

Following is the list of options that can be used.

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| Capped | Boolean | (Optional) If it is set to true, enables a capped collection. Capped collection is a fixed size collecction that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also. |
| AutoIndexID | Boolean | (Optional) If it is set to true, automatically create index on ID field. Its default value is false. |
| Size | Number | (Optional) It specifies a maximum size in bytes for a capped collection. Ifcapped is true, then you need to specify this field also. |
| Max | Number | (Optional) It specifies the maximum number of documents allowed in the capped collection. |

Let's take an **example to create collection**. In this example, we are going to create a collection name SSSIT.

1. >use test

switched to db test

1. >db.createCollection("SSSIT")

{ "ok" : 1 }

To **check the created collection**, use the command "show collections".

1. >show collections

SSSIT

How does MongoDB create collection automatically

MongoDB creates collections automatically when you insert some documents. For example: Insert a document named seomount into a collection named SSSIT. The operation will create the collection if the collection does not currently exist.

1. >db.SSSIT.**insert**({"name" : "seomount"})
2. >show collections
3. SSSIT

If you want to see the inserted document, use the find() command.

Syntax:

db.collection\_name.find()

# MongoDB Drop collection

In MongoDB, db.collection.drop() method is used to drop a collection from a database. It completely removes a collection from the database and does not leave any indexes associated with the dropped collections.

The db.collection.drop() method does not take any argument and produce an error when it is called with an argument. This method removes all the indexes associated with the dropped collection.

**Syntax:**

1. db.COLLECTION\_NAME.**drop**()

## MongoDB Drop collection example

Let's take an example to drop collection in MongoDB.

49.8M

852

History of Java

First **check the already existing collections** in your database.

1. >use mydb

Switched to db mydb

1. > show collections

SSSIT

system.indexes

**Note:** Here we have a collection named SSSIT in our database.

Now **drop the collection** with the name SSSIT:

1. >db.SSSIT.**drop**()

True

Now **check the collections** in the database:

1. >show collections

System.indexes

Now, there are no existing collections in your database.

#### **Note: The drop command returns true if it successfully drops a collection. It returns false when there is no existing collection to drop.**

MongoDB insert documents

In MongoDB, the**db.collection.insert()** method is used to add or insert new documents into a collection in your database.

**Upsert**

There are also two methods "db.collection.update()" method and "db.collection.save()" method used for the same purpose. These methods add new documents through an operation called upsert.

Upsert is an operation that performs either an update of existing document or an insert of new document if the document to modify does not exist.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

**Syntax**

1. >db.COLLECTION\_NAME.**insert**(document)

Let?s take an example to demonstrate how to insert a document into a collection. In this example we insert a document into a collection named javatpoint. This operation will automatically create a collection if the collection does not currently exist.

Example

1. db.javatpoint.**insert**(
2. {
3. course: "java",
4. details: {
5. duration: "6 months",
6. Trainer: "Sonoo jaiswal"
7. },
8. Batch: [ { **size**: "Small", qty: 15 }, { **size**: "Medium", qty: 25 } ],
9. category: "Programming language"
10. }
11. )

After the successful insertion of the document, the operation will return a WriteResult object with its status.

**Output:**

WriteResult({ "nInserted" : 1 })

Here the **nInserted** field specifies the number of documents inserted. If an error is occurred then the **WriteResult** will specify the error information.

Check the inserted documents

If the insertion is successful, you can view the inserted document by the following query.

1. >db.javatpoint.find()

You will get the inserted document in return.

**Output:**

{ "\_id" : ObjectId("56482d3e27e53d2dbc93cef8"), "course" : "java", "details" :

{ "duration" : "6 months", "Trainer" : "Sonoo jaiswal" }, "Batch" :

[ {"size" : "Small", "qty" : 15 }, { "size" : "Medium", "qty" : 25 } ],

"category" : "Programming language" }

**Note:** Here, the ObjectId value is generated by MongoDB itself. It may differ from the one shown.

MongoDB insert multiple documents

If you want to insert multiple documents in a collection, you have to pass an array of documents to the db.collection.insert() method.

Create an array of documents

Define a variable named Allcourses that hold an array of documents to insert.

1. var Allcourses =
2. [
3. {
4. Course: "Java",
5. details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" },
6. Batch: [ { **size**: "Medium", qty: 25 } ],
7. category: "Programming Language"
8. },
9. {
10. Course: ".Net",
11. details: { Duration: "6 months", Trainer: "Prashant Verma" },
12. Batch: [ { **size**: "Small", qty: 5 }, { **size**: "Medium", qty: 10 }, ],
13. category: "Programming Language"
14. },
15. {
16. Course: "Web Designing",
17. details: { Duration: "3 months", Trainer: "Rashmi Desai" },
18. Batch: [ { **size**: "Small", qty: 5 }, { **size**: "Large", qty: 10 } ],
19. category: "Programming Language"
20. }
21. ];

Inserts the documents

Pass this Allcourses array to the db.collection.insert() method to perform a bulk insert.

1. > db.javatpoint.**insert**( Allcourses );

After the successful insertion of the documents, this will return a BulkWriteResult object with the status.

BulkWriteResult({

"writeErrors" : [ ],

"writeConcernErrors" : [ ],

"nInserted" : 3,

"nUpserted" : 0,

"nMatched" : 0,

"nModified" : 0,

"nRemoved" : 0,

"upserted" : [ ]

})

**Note:**Here the nInserted field specifies the number of documents inserted. In the case of any error during the operation, the **BulkWriteResult**will specify that error.

You can check the inserted documents by using the following query:

1. >db.javatpoint.find()

Insert multiple documents with Bulk

In its latest version of MongoDB (MongoDB 2.6) provides a Bulk() API that can be used to perform multiple write operations in bulk.

You should follow these steps to insert a group of documents into a MongoDB collection.

Initialize a bulk operation builder

First initialize a bulk operation builder for the collection javatpoint.

1. var bulk = db.javatpoint.initializeUnorderedBulkOp();

This operation returns an unorder operations builder which maintains a list of operations to perform .

Add insert operations to the bulk object

1. bulk.**insert**(
2. {
3. course: "java",
4. details: {
5. duration: "6 months",
6. Trainer: "Sonoo jaiswal"
7. },
8. Batch: [ { **size**: "Small", qty: 15 }, { **size**: "Medium", qty: 25 } ],
9. category: "Programming language"
10. }
11. );

Execute the bulk operation

Call the execute() method on the bulk object to execute the operations in the list.

1. bulk.**execute**();

After the successful insertion of the documents, this method will return a **BulkWriteResult** object with its status.

BulkWriteResult({

"writeErrors" : [ ],

"writeConcernErrors" : [ ],

"nInserted" : 1,

"nUpserted" : 0,

"nMatched" : 0,

"nModified" : 0,

"nRemoved" : 0,

"upserted" : [ ]

})

Here the nInserted field specifies the number of documents inserted. In the case of any error during the operation, the **BulkWriteResult**will specify that error.

MongoDB update documents

In MongoDB, update() method is used to update or modify the existing documents of a collection.

**Syntax:**

1. db.COLLECTION\_NAME.**update**(SELECTIOIN\_CRITERIA, UPDATED\_DATA)

Example

Consider an example which has a collection name javatpoint. Insert the following documents in collection:

1. db.javatpoint.**insert**(
2. {
3. course: "java",
4. details: {
5. duration: "6 months",
6. Trainer: "Sonoo jaiswal"
7. },
8. Batch: [ { **size**: "Small", qty: 15 }, { **size**: "Medium", qty: 25 } ],
9. category: "Programming language"
10. }
11. )

After successful insertion, check the documents by following query:

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. >db.javatpoint.find()

**Output:**

{ "\_id" : ObjectId("56482d3e27e53d2dbc93cef8"), "course" : "java", "details" :

{ "duration" : "6 months", "Trainer" : "Sonoo jaiswal" }, "Batch" :

[ {"size" : "Small", "qty" : 15 }, { "size" : "Medium", "qty" : 25 } ],

"category" : "Programming language" }

**Update the existing course "java" into "android":**

1. >db.javatpoint.**update**({'course':'java'},{$**set**:{'course':'android'}})

**Check the updated document in the collection:**

1. >db.javatpoint.find()

**Output:**

{ "\_id" : ObjectId("56482d3e27e53d2dbc93cef8"), "course" : "android", "details" :

{ "duration" : "6 months", "Trainer" : "Sonoo jaiswal" }, "Batch" :

[ {"size" : "Small", "qty" : 15 }, { "size" : "Medium", "qty" : 25 } ],

"category" : "Programming language" }

# MongoDB Delete documents

In MongoDB, the db.colloction.remove() method is used to delete documents from a collection. The remove() method works on two parameters.

**1. Deletion criteria:** With the use of its syntax you can remove the documents from the collection.

**2. JustOne:** It removes only one document when set to true or 1.

**Syntax:**

1. db.collection\_name.remove (DELETION\_CRITERIA)

## Remove all documents

If you want to remove all documents from a collection, pass an empty query document {} to the remove() method. The remove() method does not remove the indexes.

Let's take an example to demonstrate the remove() method. In this example, we remove all documents from the "javatpoint" collection.

1. db.javatpoint.remove({})

## Remove all documents that match a condition

If you want to remove a document that match a specific condition, call the remove() method with the <query> parameter.

The following example will remove all documents from the javatpoint collection where the type field is equal to programming language.

1. db.javatpoint.remove( { type : "programming language" } )

## Remove a single document that match a condition

If you want to remove a single document that match a specific condition, call the remove() method with justOne parameter set to true or 1.

The following example will remove a single document from the javatpoint collection where the type field is equal to programming language.

1. db.javatpoint.remove( { type : "programming language" }, 1 )

# MongoDB Query documents

In MongoDB, the **db.collection.find()** method is used to retrieve documents from a collection. This method returns a cursor to the retrieved documents.

The db.collection.find() method reads operations in mongoDB shell and retrieves documents containing all their fields.

#### **Note: You can also restrict the fields to return in the retrieved documents by using some specific queries. For example: you can use the db.collection.findOne() method to return a single document. It works same as the db.collection.find() method with a limit of 1.**

**Syntax:**

1. db.COLLECTION\_NAME.find({})

## Select all documents in a collection:

To retrieve all documents from a collection, put the query document ({}) empty. It will be like this:

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. db.COLLECTION\_NAME.find()

**For example:** If you have a collection name "canteen" in your database which has some fields like foods, snacks, beverages, price etc. then you should use the following query to select all documents in the collection "canteen".

1. db.canteen.find()

SQL to MongoDB Mapping

The table below presents the various SQL terminology and concepts, which are similar to MongoDB terminology and concepts.

|  |  |
| --- | --- |
| **SQL Terms** | **MongoDB Terms** |
| database | Database |
| table | Collection |
| row | document or BSON document |
| column | field |
| index | index |
| table joins | $lookup, embedded document |
| primary key | primary key |
| In SQL, we can specify any unique column or column combination as the primary key. | In MongoDB, we don't need to set the primary key. The \_id field is automatically set to the primary key. |
| aggregation | aggregation pipeline |
| SELECT INTO NEW\_TABLE | $out |
| MERGE INTO TABLE | $merge |
| transactions | transactions |

**Examples below represent various SQL statements and similar MongoDB statements.**

The examples in the table assume the following conditions:

* The SQL example assumes a table name JavaTpoint
* The MongoDB examples assume a collection named JavaTpoint that contain documents of the following prototype:

1. {
2. \_id: ObjectId("509a8fb2bd2f983a0"),
3. user\_id: "admin123",
4. age: 18,
5. status: 'A'
6. }

Create and Alter commands

|  |  |
| --- | --- |
| **SQL statements** | **MongoDB statements** |
| CREATE TABLE JavaTpoint (  id MEDIUMINT NOT NULL  AUTO\_INCREMENT,  user\_id Varchar(20),  age Number,  status char(1),  PRIMARY KEY (id)  ) | db.createCollection ( " JavaTpoint " ) |
| ALTER TABLE JavaTpoint ADD join\_date DATETIME | db.JavaTpoint.updateMany(  { },  { $set: { join\_date: new Date() } }  ) |
| ALTER TABLE JavaTpoint DROP COLUMN join\_date | db.JavaTpoint.updateMany(  { },  { $unset: { "join\_date": "" } }  ) |
| CREATE INDEX idx\_user\_id\_asc ON JavaTpoint ( user\_id ) | db.people.createIndex ( { user\_id: 1 } ) |
| CREATE INDEX idx\_user\_id\_asc ON people (user\_id) | db.people.createIndex( { user\_id: 123, age: 1} ) |
| DROP TABLE people | db.people.drop () |

MongoDB and SQL Insert Statement

|  |  |
| --- | --- |
| **SQL Insert statement** | **MongoDB insert statement** |
| INSERT INTO JavaTpoint (user\_id,  age,  status)  VALUES ("mongo",  45,  "A") | db.JavaTpoint.insertOne(  { user\_id: "mongo", age: 18, status: "A" }  ) |

SQL and Mongo DB Select Command

|  |  |
| --- | --- |
| **SQL SELECT Statement** | **MongoDB find() Statement** |
| SELECT \*  FROM JavaTpoint | db.JavaTpoint.find() |
| SELECT id, user\_id, status FROM JavaTpoint | db.JavaTpoint.find( { }, { user\_id: 1, status: 1 } ) |
| SELECT user\_id, status FROM JavaTpoint | db.JavaTpoint.find( { }, { user\_id: 1, status: 1, \_id: 0 } ) |
| SELECT \* FROM JavaTpoint WHERE status = "B" | db.JavaTpoint.find( { status: "A" } ) |
| SELECT user\_id, status FROM JavaTpoint WHERE status = "A" | db.javaTpoint.find( { status: "A" }, { user\_id: 1, status: 1, \_id: 0 } ) |
| SELECT \* FROM JavaTpoint WHERE status != "A" | db.JavaTpoint.find( { status: { $ne: "A" } } ) |
| SELECT \*  FROM JavaTpoint  WHERE status = "A"  AND age = 50 | db.JavaTpoint.find(  { status: "A",  age: 50 }  ) |
| SELECT \*  FROM JavaTpoint  WHERE status = "A"  OR age = 50 | db.JavaTpoint.find(  { $or: [ { status: "A" } , { age: 50 } ] }  ) |
| SELECT \*  FROM JavaTpoint  WHERE age > 25 | db.JavaTpoint.find(  { age: { $gt: 25 } }  ) |
| SELECT \*  FROM JavaTpoint  WHERE age < 25 | Db.JavaTpoint.find(  { age: { $lt: 25 } }  ) |
| SELECT \*  FROM JavaTpoint  WHERE age > 25  AND age <= 50 | db.JavaTpoint.find(  { age: { $gt: 25, $lte: 50 } }  ) |
| SELECT \*  FROM JavaTpoint  WHERE user\_id like "%bc%" | db.JavaTpoint.find( { user\_id: /bc/ } )  -or-  db.JavaTpoint.find( { user\_id: { $regex: /bc/ } } ) |
| SELECT \*  FROM JavaTpoint  WHERE user\_id like "bc%" | db.JavaTpoint.find( { user\_id: /^bc/ } )  -or-  db.JavaTpoint.find( { user\_id: { $regex: /^bc/ } } ) |
| SELECT \*  FROM JavaTPoint  WHERE status = "A"  ORDER BY user\_id ASC | db. JavaTPoint. find( { status: "A" } ). sort( { user\_id: 1 } ) |
| SELECT \*  FROM JavaTPoint  WHERE status = "A"  ORDER BY user\_id ASC | db. JavaTPoint. find( { status: "A" } ). sort( { user\_id: 1 } ) |
| SELECT \*  FROM JavaTPoint  WHERE status = "A"  ORDER BY user\_id ASC | db. JavaTPoint. find( { status: "A" } ). sort( { user\_id: 1 } ) |
| SELECT \*  FROM JavaTPoint  WHERE status = "A"  ORDER BY user\_id DESC | db. JavaTPoint. find( { status: "A" } ). sort( { user\_id: -1 } ) |
| SELECT \*  FROM JavaTPoint  WHERE status = "A"  ORDER BY user\_id DESC | db. JavaTPoint. find( { status: "A" } ). sort( { user\_id: -1 } ) |
| SELECT COUNT(\*)  FROM JavaTPoint | db. JavaTPoint. count()  or  db. JavaTPoint. find(). count() |
| SELECT COUNT(user\_id)  FROM JavaTPoint | db. JavaTPoint.count( { user\_id: { $exists: true } } )  or  db. JavaTPoint.find( { user\_id: { $exists: true } } ).count() |
| SELECT COUNT(\*)  FROM JavaTPoint  WHERE age > 30 | db. JavaTPoint.count( { age: { $gt: 30 } } )  or  db. JavaTPoint.find( { age: { $gt: 30 } } ).count() |
| SELECT DISTINCT(status)  FROM JavaTPoint | db. JavaTPoint.aggregate( [ { $group : { \_id : "$status" } } ] )  or, for distinct value sets that do not exceed the BSON size limit  db. JavaTPoint.distinct( "status" ) |
| SELECT \*  FROM JavaTPoint  LIMIT 1 | db. JavaTPoint.findOne()  or  db. JavaTPoint.find(). limit(1) |
| SELECT \*  FROM JavaTPoint  LIMIT 5  SKIP 10 | db. JavaTPoint.find(). limit(5). skip(10) |
| EXPLAIN SELECT \*  FROM JavaTPoint WHERE status = "A" | db. JavaTPoint. find( { status: "A" } ). explain() |

SQL and MongoDB Update Statements

|  |  |
| --- | --- |
| **SQL Update Statements** | **MongoDB updateMany() Statements** |
| UPDATE JavaTpoint SET status = "C"  WHERE age > 25 | db.JavaTpoint.updateMany( { age: { $gt: 25 } }, { $set: { status: "C" } } ) |
| UPDATE JavaTpoint SET age = age + 3  WHERE status = "A" | db.JavaTpoint.updateMany( { status: "A" } , { $inc: { age: 3 } } ) |

SQL and MongoDB Delete Statements

|  |  |
| --- | --- |
| **SQL Delete Statements** | **MongoDB deleteMany() Statements** |
| DELETE FROM JavaTpoint WHERE status = "D" | db.JavaTpoint.deleteMany( { status: "D" } ) |
| DELETE FROM JavaTpoint | db.JavaTpoint.deleteMany( { } ) |

MongoDB text search

MongoDB performs a text search of string content using the query operation. It uses a text index and operator to perform the text search.

Example:

It shows you "how to build a text index and use it to find your books".

Now, you have to create a collection named "***library***" as follows:

1. db.stores.insert(
2. [
3. { \_id: 101, name: "Java", description: "By ABC" },
4. { \_id: 102, name: "MongoDB", description: "By XYZ" },
5. { \_id: 103, name: "Python", description: "By ABCD" },
6. { \_id: 104, name: "Engineering Mathematics", description: "By \*\*\*\*\*" },
7. { \_id: 105, name: "Salesforce", description: "By Salesforce" }
8. ]
9. )

Text Index

In MongoDB we have text indexes to support text search queries on string content. The field that have any string value or an array of string elements may include by text indexes

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

We must have a text index in our collection to perform text search queries. In a table/collection, we can have only one text search index. But multiple fields can be covered by a single index.

We can run the following example in Mongo shell to allow text search cover the name and description fields:

1. db.library.createIndex( { name: "text", description: "text" } )

$text Operator

We can use the $text operator to perform text searches on a table with a text index. The $text operator will flag the search string which uses the whitespaces and also most of the punctuation as delimiters. The $text operator performs a logical OR operation for all such tokens in the search string.

In the below example, we can use the query to find all libraries containing any books name related to "MongoDB", "Java", "DBMS", etc.

db.library.find( { $text: { $search: "Java" } } )

Using the $text operator, we can also search for exact phrases by wrapping them in double-quotes. Only those documents will be matched that include the phrases.

**For Example -**

1. db.library.find( { $text: { $search: "\"Java Book\"" } } )

Sorting

MongoDB return the result by default in unsorted order. An optimum score will be computed for each document by the text search query that specifies how well a document matches the query.

1. db.library.find(
2. { $text: { $search: "java" } },
3. { score: { $meta: "textScore" } }
4. ).sort( { score: { $meta: "textScore" } } )

In the above example we explicitly project the meta textScore field to sort the result in order of relevance score.

# MongoDB Shell

MongoDB have a JavaScript shell that allows interaction with MongoDB instance from the command line.

If you want to create a table, you should name the table and define its column and each column's data type.

The shell is useful for performing administrative functions and running instances.

## How to run the shell

To start the shell, open command prompt, run it as a administrator then run the mongo executable:

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. $ mongo

MongoDB shell version: 2.4.0

Connecting to: test

You should start [mongoDB](https://www.javatpoint.com/mongodb-tutorial) before starting the shell because shell automatically attempt to connect to a MongoDB server on startup.

The shell is a full-featured JavaScript interpreter. It is capable of running Arbitrary [JavaScript](https://www.javatpoint.com/javascript-tutorial) program.

**Let us take a simple mathematical program:**

1. >x= 100
2. 100
3. >x/ 5;
4. 20

**You can also use the JavaScript libraries**

1. > "Hello, World!".replace("World", "MongoDB");

Hello, MongoDB!

**You can even define and call JavaScript functions**

1. > **function** factorial (n) {
3. ... if (n <= 1) **return** 1;
5. ... **return** n \* factorial(n - 1);
6. ... }
8. > factorial (5);
10. 120

#### **Note: You can create multiple commands.**

When you press "Enter", the shell detect whether the JavaScript statement is complete or not.

If the statement is not completed, the shell allows you to continue writing it on the next line. If you press "Enter" three times in a row, it will cancel the half-formed command and get you back to the > - prompt.

MongoDB Shell Collection Methods

Following are the MongoDB collection methods that are used in different scenarios.

#1: db.collection.aggregate(pipeline, option)

The aggregate method calculates mass values for the data in a collection/table or in a view.

**Pipeline:** It is an array of mass data operations or stages. It can accept the pipeline as a separate argument, not as an element in an array. If the pipeline is not specified as an array, then the second parameter will not be specified.

**Option:** A document that passes the aggregate command. It will be available only when you specify the pipeline as an array.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

**Command Fields:**

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| explain | boolean | The explain field specifies to return the information on the processing of the pipeline. |
| allowDiskUse | boolean | The allow disk use field enables you to write to temporary files. |
| cursor | document | The initial batch size for the cursor is specified with this field. The value inside this field is the document with the batchSize field. |
| maxTimeMS | non-negative integer | Use the time limit for the processing operations on a cursor is specified using this field. |
| bypassDocument | Validation | boolean The $out or $merge aggregation stages can be specified using this field. It allows the aggregate collection method to bypass the document validation during the operation. |
| readConcern | document | You can specify the read concern level using this field. |
| collation | document | The collation field specifies the language specific rules for string comparison. |

**Example**

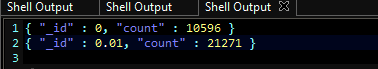
The examples use the collection library that contains the following documents:

1. { \_id: 1, book\_id: "Java", ord\_date: ISODate("2012-11-02T17:04:11.102Z"), status: "A", amount: 50 }
2. { \_id: 0, book\_id: "MongoDB", ord\_date: ISODate("2013-10-01T17:04:11.102Z"), status: "A", amount: 100 }
3. { \_id: 0.01, book\_id: "DBMS", ord\_date: ISODate("2013-10-12T17:04:11.102Z"), status: "D", amount: 25 }
4. { \_id: 2, book\_id: "Python", ord\_date: ISODate("2013-10-11T17:04:11.102Z"), status: "D", amount: 125 }
5. { \_id: 0.02, book\_id: "SQL", ord\_date: ISODate("2013-11-12T17:04:11.102Z"), status: "A", amount: 25 }

**Calculating the sum**

1. db.library.aggregate([
2. { $match: { status: "A" } },
3. { $group: { \_id: "$book\_id", total: { $count: "$amount" } } },
4. { $sort: { total: -1 } }
5. ])

**Output:**



**Specifying the collation**

1. db.library.aggregate(
2. [ { $match: { status: "A" } }, { $group: { \_id: "$ord\_date", count: { $count: 1 } } } ],
3. { library: { locale: "fr", strength: 1 } } );

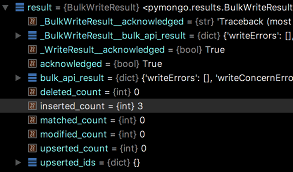
#2 db.collection.bulkWrite()

The bulkWrite() method performs multiple write operations with the order of execution control. Array of write operations are executed by this operation. Operations are executed in a specific order by default.

**Syntax:**

1. db.collection.bulkWrite(
2. [ **<op.** 1**>**, **<op.** 2**>**, .. ],
3. {
4. writeConcern : **<document>**,
5. ordered: **<boolean>**
6. }
7. )

**Output:**



Execution of Operations

**insertOne:** It inserts only one document into the collection.

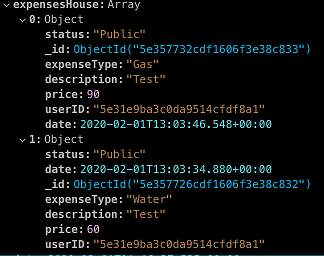
1. db.collection.bulkWrite( [
2. { insertOne : { "document" : **<document>** } }
3. ] )



**Update one:** It updates only one document that matches the filter in the collection.

1. db.collection.bulkWrite( [
2. { updateOne :
3. {
4. "filter": **<document>**,
5. "update": **<document** or pipeline**>**,
6. "upsert": **<boolean>**,
7. "collation": **<document>**,
8. "arrayFilters": [ **<filterdocument1>**, ... ],
9. "hint": **<document**|string**>**
10. }
11. }
12. ] )

**Output:**



**Update Many:** It updates all the filter matched documents in the collection.

1. db.collection.bulkWrite( [
2. { updateMany :{
3. "filter" : **<doc.>**,
4. "update" : **<document** or pipeline**>**,
5. "upsert" : **<Boolean>**,
6. "collation": **<document>**,
7. "arrayFilters": [ **<filterdocument1>**, ... ],
8. "hint": **<document**|string**>**                   // Available starting in 4.2.1
9. }
10. }
11. ] )

**replaceOne:** It replaces a single document in the collection that matches the filter.

1. db.collection.bulkWrite([
2. { replaceOne :
3. {
4. "filter" : **<doc.>**,
5. "replacement" : **<doc.>**,
6. "upsert" : **<boolean>**,
7. "collation": **<document>**,
8. "hint": **<document**|string**>**
9. }
10. }
11. ] )

# 3. db.collection.count(query, option)

The count() method return the number of documents that would match a find method query for the collection or view.

**Example:**

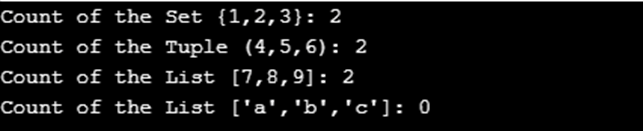
We will count all the document in the javaTpoint collection using the following operation:

1. db.javaTpoint.count()

Now, we will count all the documents that Match a Query in the javaTpoint collection with the field tut\_dt greater than new Date ('01/01/2015')

1. db.javaTpoint.count( { tut\_dt: { $gt: new Date('01/01/2015') } } )

**Output:**



#4. Db.collection.countDocuments(query, options)

The countDocument() method return the number of documents that match the query for a collection or view. it does not use the metadata to return the count.

**Syntax:**

1. db.collection.countDocuments( **<query>**, **<options>** )

**Examples:**

Below example will count the number of all the documents in the javaTpoint collection.

1. db.javaTpoint.countDocuments({})

Now, we will count all the documents that Match a Query in the javaTpoint collection with the field tut\_dt greater than new Date ('01/01/2015')

1. db.javaTpoint.countDocuments( { tut\_dt: { $gt: new Date('01/01/2015') } } )

#5. db.collection.estimatedDocumentCount()

The estimateddocumentCount() method counts all documents in a collection or view. This method wraps the count command.

**Syntax:**

1. db.collection.estimatedDocumentCount( **<options>** )

**Example**

The following example will retrieve the count of all the documents in the [javaTpoint](https://www.javatpoint.com/) collection:

1. db.javaTpoint.estimatedDocumentCount({})

#6. db.collection.createIndex()

It can create the indexes on collections

**Syntax:**

1. db.collection.createIndex(keys, options)

*Keys:*For an ascending index on a field we need to specify a value of 1 and for the descending index we need to specify a value of -1.

**Example**

The example below creates an ascending index on the field tut\_Date.

1. db.collection.createIndex( { tut\_Date: 1 } )

The following example shows a compound index created on the tut\_Date field and the tut\_code field.

1. db.collection.createIndex( { tut\_Date: 1, tut\_code: -1 } )

The example below will create an index named as category\_tutorial. The example creates the index with the collation that specifies the locale fr and comparison strength.

1. db.collection.createIndex(
2. { category: 1 },
3. { name: "category\_tutorial", collation: { locale: "fr", strength: 2 } }
4. )

#7. db.collection.createIndexes()

The createIndexes() method creates one or more indexes on a collection.

**Syntax:**

1. db.collection.createIndexes( [keyPatterns, ]options)

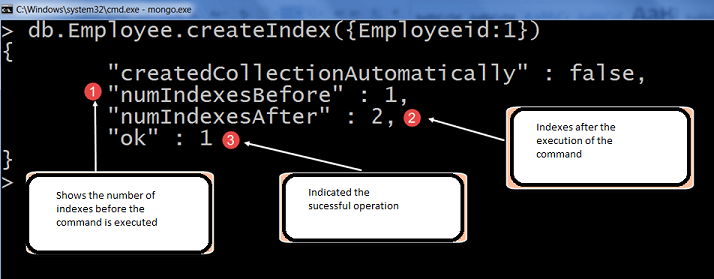
*Keypatterns*: It is an array that contains the index specific documents. All the document have field-value pairs. For an ascending index on a field we need to specify a value of 1 and for the descending index we need to specify a value of -1

**Example**

In the example below we considered a employee collection containing documents that resemble the following:

1. {
2. location: {
3. type: "Point",
4. coordinates: [-73.8577, 40.8447]
5. },
6. name: "Employee",
7. company: "Amazon",
8. borough: "CA",
9. }

**Output:**



Now, the example below creates two indexes on the products collection:

* Index on the manufacturer field in ascending order.
* Index on the category field in ascending order.

The above indexes use a collation which specifies the basic fr and comparison strength  as 2.

1. db.products.createIndexes( [ { "manufacturer": 1}, { "category": 1 } ],
2. { collation: { locale: "fr", strength: 2 } })

#8. db.collection.dataSize()

The data size method have a cover around the output of the collStats (i.e. db.collection.stats() ) command.

#9. db.collection.deleteOne()

The deleteOne() method deletes one document from the collection. It replaces the first document that is similar to the filter. You need to use a field that is related to a unique index such as id for perfect deletions.

**Syntax:**

1. db.collection.deleteOne(
2. **<filter>**,
3. {
4. writeConcern: **<document>**,
5. collation: **<document>**
6. }
7. )

**Example**

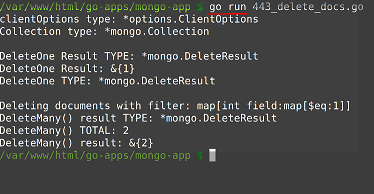
The orders collection has documents with the following structure:

1. {
2. \_id: objectId("563237a41a4d6859da"),
3. book: "",
4. qty: 2,
5. type: "buy-limit",
6. limit: 10,
7. creationts: ISODate("2015-11-01T2:30:15Z"),
8. expiryts: ISODate("2015-11-01T2:35:15Z"),
9. client: "JavaTpoint"
10. }

The following operation deletes the order with \_id: objectId ("563237a41a4d6858 2da"):

1. try {
2. db.orders.deleteOne( { "\_id" : objectId("563237a41a4d68582da") } );
3. } catch (e) {
4. print(e);
5. }

**Output:**



MongoDB Cursor Methods

The MongoDB cursor methods modifies the way that the specified query is executed. Following are the list of the cursor methods with description, syntax, and examples.

#1 cursor.addOption(flag)

The method adds "OP\_QUERY" wire protocol flags. It is added to change the behaviour of queries like tailaible flag.

**Example**

1. var t = db.myCappedCollection;
2. var cursor = t.find().
3. addOption(DBQuery.Option.tailable)
4. .addOption(DBQuery.Option.awaitData)

The above example adds the tailable flag and the awaitData flag to ensure that the query returns a tailable cursor. A cursor will be generated using this method that waits for few seconds after returning the full result set. So that during the query it can get and returns the additional data.

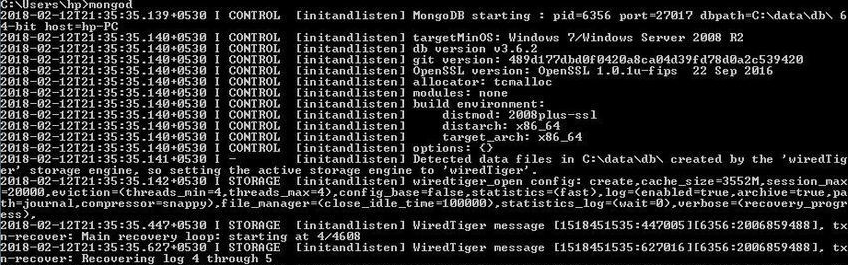
Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

#2. Cursor.batchSize(size)

The batch result from the [MongoDB](https://www.javatpoint.com/mongodb-tutorial) object returns the number of documents which is specified using the batch size method. In many cases, if we modify the batch size, it will not be going to affect the user or the application.

**Example**

1. db.inventory.find().batchSize(10)



#3. cursor.close()

The method used to close the cursor and release the associated server resources on the instruction of the method. The cursor will be automatically closed by the server that have zero remaining results or it have been idle for a specified period of time.

**Example**

1. db.collection.find(**<query>**).close()

#4. cursor.collation(<collation document>)

The MongoDB collation() method specifies the collation for the cursor returned by the db.collection.find().

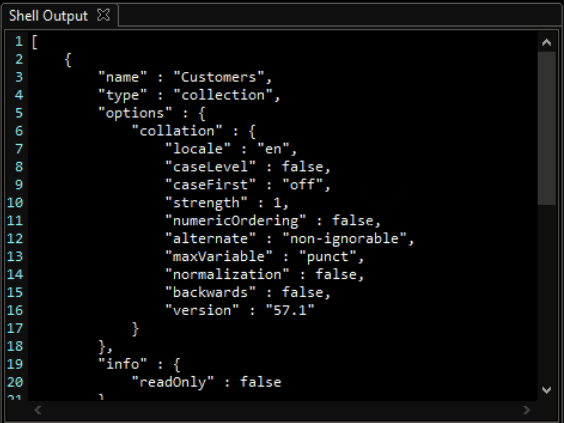
Collation documents that is accepted by the close method:

1. {
2. locale: **<string>**,
3. caseLevel: **<boolean>**,
4. caseFirst: **<string>**,
5. strength: **<int>**,
6. numericOrdering: **<boolean>**,
7. alternate: **<string>**,
8. maxVariable: **<string>**,
9. backwards: **<boolean>**
10. }

**Example:**

1. db.javaTpoint.find( { x: "a" } ).collation( { locale: "en\_US", strength: 1 } )

**Output:**



#5. cursor.forEach(function)

JavaScript function will be applied to all the documents by the cursor using the forEach method.

**Syntax:**

1. db.collection.find().forEach(**<function>**)

**Example:**

The forEach() method invoked on the cursor returned by find() method to display the name of all the users in the collection:

1. db.users.find().forEach( function(javaTpoint) { print( "user: " + editors.name ); } );

**Output:**



#6. cursor.hint(index)

The method is called during the query to override the MongoDB's default selection of index and query optimization process.

**Examples:**

All documents in the user's collection using the index on the age field will be returned using the query below.

1. db.users.find().hint( { age: 1 } )

#7. cursor.limit()

This method is used to specify the maximum number of documents returned by the cursor. It will be used within the cursor and comparable to the LIMIT statement in a SQL database.

**Example:**

1. db.collection.find(**<query>**).limit(**<number>**)

#8. cursor.map(function)

The map method is used by the document visited by the cursor and also collects the return values from nearest application into an array.

**Example:**

1. db.users.find().map( function(u) { return u.name; } );

#9. cursor.max()

The max method is used to restrict the results of find().max() method. MongoDB specifies the exclusive upper bound for a specific index that provides a way to specify an upper bound for the compound key indexes.

Example

Create the following indexes for the collection:

1. db.products.createIndexes( [
2. { "item" : 1, "type" : 1 },
3. { "item" : 1, "type" : -1 },
4. { "price" : 1 }])

If you were using the order of { item: 1, type: 1 } index, the max() restricts the query of the documents that are below the bound of item equal to Mango and type equal to

1. db.products.find().max( { item: 'Mango', type: 'Alfonso' } ).hint( { item: 1, type: 1 } )

#10. cursor.min()

To constrain the results of find(). min() MongoDB specifies the lower bound for a specific index in order . This method provides a way to define the lower bounds on compound key indexes.

**Syntax:**

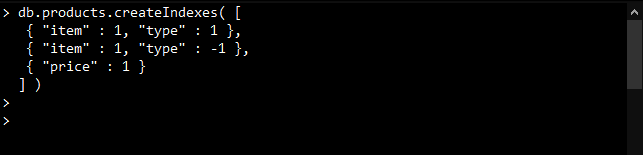
1. { field1: **<min** value**>**, field2: **<min** value2**>**, fieldN:**<min** valueN**>** }

**Example:**

First, create a sample collection named as superstore that holds the following documents:

1. db.products.insertMany([
2. { "\_id" : 1, "item" : "Java", "type" : "book", "price" : NumberDecimal("1.09") },
3. { "\_id" : 2, "item" : "MongoDB", "type" : "book", "price" : NumberDecimal("1.9") },
4. { "\_id" : 3, "item" : "Homefoil","type" : "Utensil", "price" : NumberDecimal("1.2") },
5. { "\_id" : 4, "item" : "Handwash", "type": "Utensil", "price" : NumberDecimal("1.29") },
6. { "\_id" : 5, "item" : "Rice", "type" : "Grocery", "price" : NumberDecimal("1.59") },
7. { "\_id" : 6, "item" : "Mango", "type" : "Fruit", "price" : NumberDecimal("1.29") },
8. { "\_id" : 7, "item" : "Orange", "type" : "Fruit", "price" : NumberDecimal("2.99") },
9. { "\_id" : 9, "item" : "Apple", "type" : "Fruit", "price" : NumberDecimal("1.99") },
10. { "\_id" : 8, "item" : "Potato", "type" : "vegetable", "price" : NumberDecimal("0.99") },
11. { "\_id" : 10, "item" : "Onion", "type" : "vegetable", "price" : NumberDecimal("1.39") }
12. ])

Now, create the indexes for the collection:



The min() method limits the query to the documents using the ordering of the { item: 1, type: 1 } index.

1. db.products.find().min( { item: 'Java', type: 'book' } ).hint( { item: 1, type: 1 } )

#11. cursor.tailable()

The tailable method marks the cursor as tailable. It works as a scanner over a capped collection. Even after reaching the last node of the collection it remain open. The application of the method will be continuously running as new data inserted inside the collection.

**Syntax:**

1. cursor.tailable( { awaitData : **<boolean>** } )

If the awaitdata flag is set to true, MongoDB blocks the query thread for a period of time when the end of the capped collection was reached by the cursor waiting for new data to arrive. The blocked thread is signaled to wake up and return the next batch to the client when new data is inserted into the capped collection.

#12. cursor.toArray()

The method returns an array that have all the documents belongs to the cursor. It loads all the documents into RAM and exhaust the cursor by iterating the cursor completely.

**Example:**

1. var allProductsArray = db.products.find().toArray();
2. if (allProductsArray.length **>** 0) { printjson (allProductsArray[0]); }

MongoDB Database commands

The MongoDB database commands are used to create, modify, and update the database.

#1. db.adminCommand(cmd)

The admin command method runs against the admin database to run specified database commands by providing a helper.

Command: Either the argument is specified in the document form or a string form. If the command is defined as a string, it cannot include any argument.

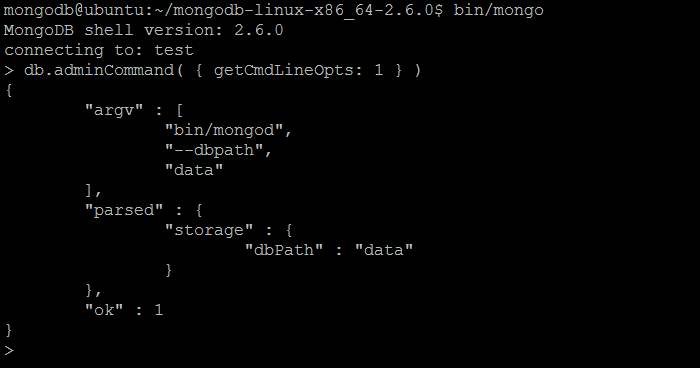
**Example:**

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

Creating a user named JavaTpoint with the dbOwner role on the admin database.

1. db.adminCommand(
2. {
3. createUser: "JavaTpoint",
4. pwd: passwordPrompt(),
5. roles: [
6. { role: "dbOwner", db: "admin" }
7. ]
8. }
9. )

**Output:**



#2. db.aggregate()

The aggregate method initialize a specific diagnostic or admin pipeline, which does not require anu underlying collection.

**Syntax:**

1. db.aggregate( [ **<pipeline>** ], { **<options>** } )

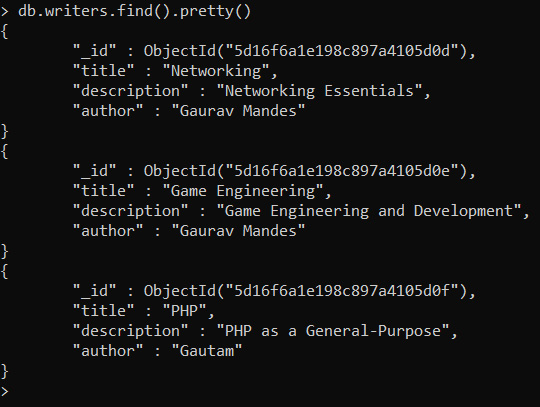
The pipeline parameter does not require any underlying collection and always starts with a compatible stage, such as $currentOp or $listLocalSessions. It is an array of stages that will be executed.

**Example:**

The following example runs a pipeline with two stages. The first is the $currentOp operation and the second will filters the results.

1. use admin
2. db.aggregate( [ {
3. $currentOp : { allUsers: true, idleConnections: true } },
4. {
5. $match : { shard: "shardDemo" }
6. }
7. ] )

**Output:**



#3. db.cloneDatabase("hostname")

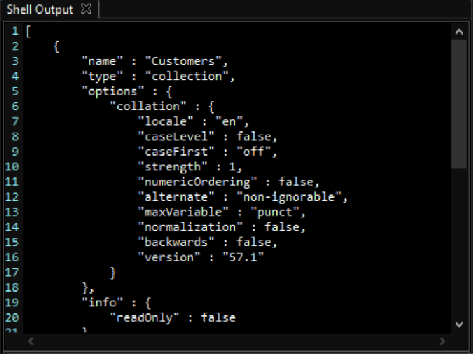
The clonedatabase method copies the specified database to the current database and assumes that the database at the remote location has the same name as the current database.

The hostname parameter contains the hostname of the database that we want to copy.

**Example:**

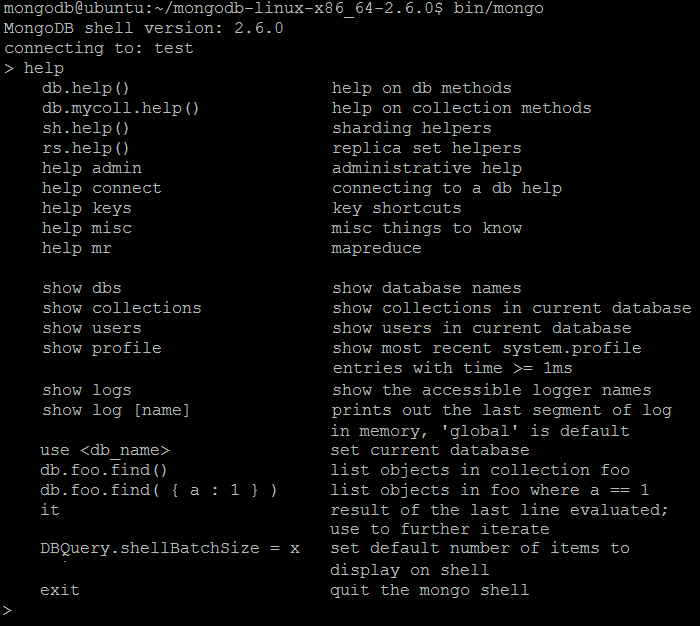
db.cloneDatabase("customers")

**Output:**



#4. db.commandHelp(command)

We have the help option for the specified database command using the commandHelp method. The command parameter contains the name of a database command.

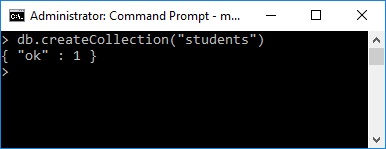


#5. db.createCollection(name, options)

A new collection or view will be created using this method. The createCollection method is used primarily for creating new collections that use specific options when the collection is first referenced in a command.

**For example** - we will create a [javaTpoint](https://www.javatpoint.com/) collection with a [JSON](https://www.javatpoint.com/json-tutorial) Schema validator:

1. db.createCollection( "student", {
2. validator: { $jsonSchema: {
3. bsonType: "object",
4. required: [ "phone" ],
5. properties: {
6. phone: {
7. bsonType: "string",
8. description: "must be a string and is required"
9. },
10. email: {
11. bsonType : "string",
12. pattern: "@mongodb\.com$",
13. description: "must be a string and match the regular expression pattern"
14. },
15. status: {
16. enum: [ "Unknown", "Incomplete" ],
17. description: "can only be one of the enum values"
18. }
19. }
20. } }
21. } )



#6. db.createView()

When we applying the specified aggregation pipeline to the collection, the createView method create a new view for the collection. The method can be computed during the read operations and acts as read-only operations. The views can be created in the same database of the source collection to executes read operations as a part of the underlying aggregation pipeline.

**Syntax:**

1. db.createView(**<view>**, **<source>**, **<pipeline>**, **<options>**)

The following example creates a StudentFeedback view with the \_id, student.management, and department fields:

1. db.createView(
2. "StudentFeedback",
3. "survey",
4. [ { $project: { "management": "$Student.management", department: 1 } } ]
5. )

**Output:**



#7. db.dropDatabase(<writeConcern>)

The drop method removes the specified database and the associated data files.

**For example -**

we use <database> operation to switch the current database to the temporary database. We use the db.dropDatabase() method to drops the temporary database

1. use temp
2. db.dropDatabase()

#8. db.getLogComponents()

The getLog method returns the current stiltedly settings. The method determines the amount of Log Messages produced by [MongoDB](https://www.javatpoint.com/mongodb-tutorial) for each log message component.

**Example:**

1. {
2. "verbosity" : 0,
3. "accessControl" : {
4. "verbosity" : -1
5. },
6. "command" : {
7. "verbosity" : -1
8. },
9. "control" : {
10. "verbosity" : -1
11. },
12. "geo" : {
13. "verbosity" : -1
14. },
15. "index" : {
16. "verbosity" : -1
17. },
18. "network" : {
19. "verbosity" : -1
20. },
21. "query" : {
22. "verbosity" : 2
23. },
24. "replication" : {
25. "verbosity" : -1,
26. "election" : {
27. "verbosity" : -1
28. },
29. "heartbeats" : {
30. "verbosity" : -1
31. },
32. "initialSync" : {
33. "verbosity" : -1
34. },
35. "rollback" : {
36. "verbosity" : -1
37. }
38. },
39. "sharding" : {
40. "verbosity" : -1
41. },
42. "storage" : {
43. "verbosity" : 2,
44. "recovery" : {
45. "verbosity" : -1
46. },
47. "journal" : {
48. "verbosity" : -1
49. }
50. },
51. "write" : {
52. "verbosity" : -1
53. }
54. }

MongoDB Query Plan Cache Methods

#1. db.collection.getPlanCache()

Using the plan cache object of a collection allows you to access this method. We need to use db.collection.getPlanCache() method to retrieve the plan cache object. The method returns an interface that has been used for collection to access the query plan cache and provides a method to view and clear the query plan cache.

The query shapes can be cached by the query optimizer that has more than one implementable plan.

**The following methods are available through the interface:**

**PlanCache.help:** This method is accessible through the plan cache object of the specified collection an displays the methods available for the specified collection's query plan cache, *i.e., db.collection.getPlanCache().help().*

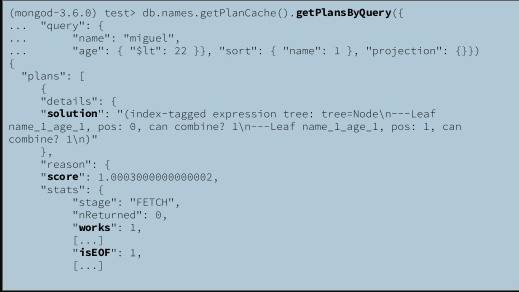
Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

**PlanCache.listQueryShapes()**: This method is accessible through the plan cache object of the specified collection and displays the query shapes for which cached query plan exists, *i.e. db.collection.getPlanCache().listQueryShapes().*

**PlanCache.getPlansByQuery()**: This method is accessible through the plan cache object of the specified collection and displays the cached query plans for the argumented query shape., *i.e. db.collection.getPlanCache().getPlansByQuery().*

**PlanCache.clearPlansByQuery()**: This method is accessible through the plan cache object of a specific collection and clears the cached query plans for the argumented query shape., *i.e. db.collection.getPlanCache().clearPlansByQuery()*

**PlanCache.clear()**: This method is accessible through the plan cache object of a specific collection and clears all the cached query plans for a collection., *i.e. db.collection.getPlanCache().clear()*



#2. PlanCache.clear()

The clear method is only available from the objects of plan cache to remove all cached query plans for the specified collection.

**For example -** Clearing the cache for the student collection.

1. db.student.getPlanCache().clear()



#3. PlanCache.clearPlansByQuery()

This method clears the cached query plans for the specified query shape and available from the plan cache object of a specific collection; *i.e.*

1. db.collection.getPlanCache().clearPlansByQuery( **<query>**, **<projection>**, **<sort>** )

The PlanCache.clearPlansByQuery() method accepts the following parameters:

**Query document:** This parameter is the query predicate of the query shape. The structure of the predicate and the field names are significant to the shape and the values in the query predicate are insignificant.

**Projection document:** It is the optional parameter, required if specifying the sort parameter. It can be used for the projection associated with the query shape.

**Example**

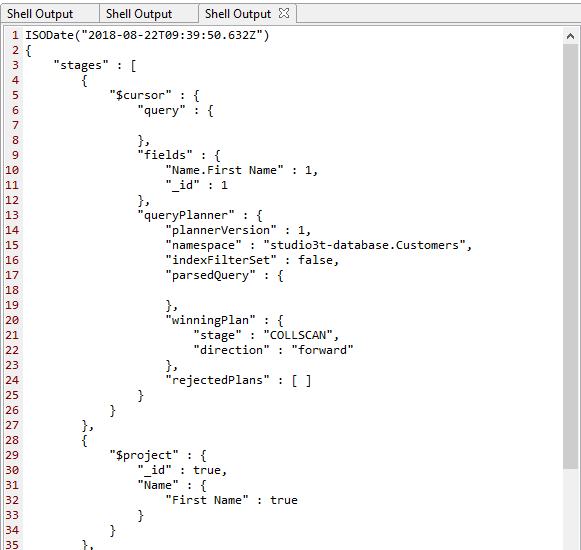
If a collection orders has the following query shape:

1. {
2. "query" : { "qty" : { "$gt" : 10 } },
3. "sort" : { "ord\_date" : 1 },
4. "projection" : { },
5. "queryHash" : "9AAD95BE" // Available starting in MongoDB 4.2
6. }

The following operation removes the query plan cached for the shape:

1. db.orders.getPlanCache().clearPlansByQuery(
2. { "qty" : { "$gt" : 10 } },
3. { },
4. { "ord\_date" : 1 }
5. )

**Output:**



#4. PlanCache.help()

Displays the methods available to view and modify a collection's query plan cache. The method is only available from the plan cache object of a specific collection; i.e.

1. db.collection.getPlanCache().help()

#5. PlanCache.listQueryShapes()

The method is deprecated from the version 4.2 and was used to display the query shapes for which the cached query plans exist. You can use this method if you are using the older version of MongoDB. All the query shapes are associated with query hash to help in identification of the queries that are slow with the same query shape.

**Syntax:**

1. db.collection.getPlanCache().listQueryShapes()

**Example:**

1. db.orders.getPlanCache().listQueryShapes()

The order collection has the cashed plan associated with the query shape. The method will return an array of the query shape that is currently inside the cache.

1. [
2. {
3. "query" : { "qty" : { "$gt" : 10 } },
4. "sort" : { "ord\_date" : 1 },
5. "projection" : { },
6. "queryHash" : "9AAD95BE"
7. {
8. "query" : { "$or" :
9. [
10. { "qty" : { "$gt" : 15 }, "item" : "xyz123" },
11. { "status" : "A" }
12. ]
13. },
14. "sort" : { },
15. "projection" : { },
16. "queryHash" : "0A087AD0"
17. },
18. {
19. "query" : { "$or" : [ { "qty" : { "$gt" : 15 } }, { "status" : "A" } ] },
20. "sort" : { },
21. "projection" : { },
22. "queryHash" : "DA43B020"
23. }
24. ]

MongoDB User Management Methods

The MongoDB user management methods are used to manage the user of the database.

#1. db.auth()

The auth method allows a user within the shell to provide authentication to the database. It can accept either the user name and password, i.e, db.auth( <username>, passwordPrompt() ) Or db.auth( <username>, <password> ).

We can define a user collection that contains the name of the user, password, the mechanism, and a digest password flag.

1. db.auth( {
2. user: **<name>**,
3. pwd: "**<cleartext** password**>**",
4. mechanism: **<auth.** mechanism**>**,
5. digestPassword: **<boolean>**
6. } )

**Example:**

After connecting the mongo shell, if you want to authenticate, you have to issue db.auth() in the user's authentication database:

1. use test
2. db.auth( "javaTpoint", passwordPrompt() )



#2. db.changeUserPassword(username, password)

Updates a user's password. Run the method in the database where the user is defined, i.e. the database you created the user.

**Example**

The following operation changes the password of the user named accountUser in the products database to SOh3TbYhx8ypJPxmt1oOfL:

1. use products
2. db.changeUserPassword("accountUser", passwordPrompt())

You can also pass the new password directly to db.changeUserPassword():

1. use products
2. db.changeUserPassword("accountUser", "SOh3TbYhx8ypJPxmt1oOfL")

**Output:**



#3. db.createUser(<user>, <writeConcern>)

This method creates a new user that is specified in the argument for the current database on which the method is currently running. In the case user already exists on the specified database the method return a duplication error.

Syntax to defines the user of the database using createUser method:

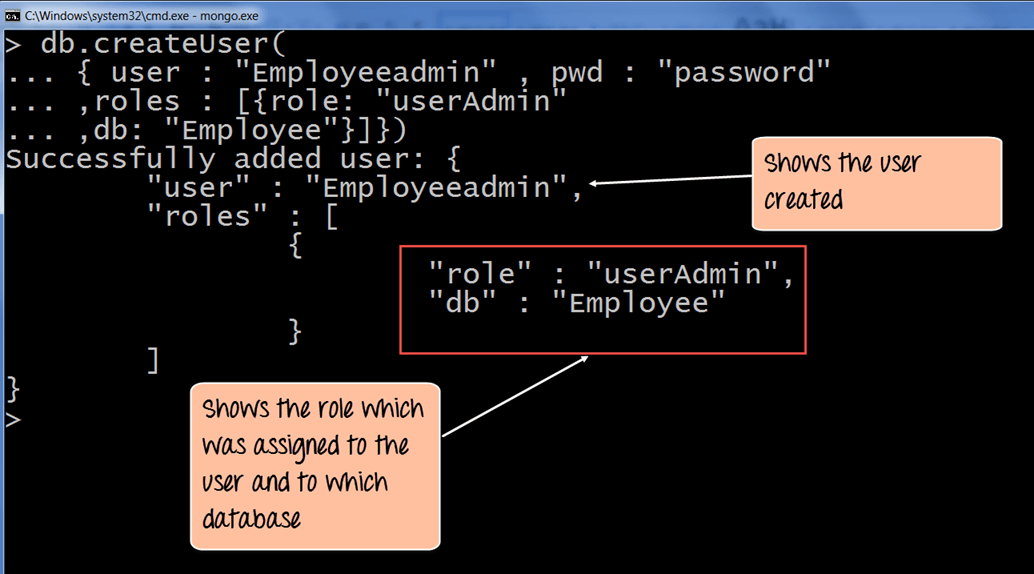
1. {
2. user: "**<name>**",
3. pwd: "**<cleartext** password**>**",
4. customData: { **<any** info.**>** },
5. roles: [
6. { role: "**<role>**", db: "**<database>**" } | "**<role>**",
7. ...
8. ],
9. authenticationRestrictions: [
10. {
11. clientSource: ["**<IP>**" | "**<CIDR** range**>**", ...],
12. serverAddress: ["**<IP>**" | "**<CIDR** range**>**", ...]
13. } ]
14. mechanisms: [ "**<SCRAM-SHA-1**|SCRAM-SHA-256**>**", ... ],
15. passwordDigestor: "**<server**|client**>**"
16. }

**Examples:**

The following example will create the accountJTP user on the student database.

1. use EmployeeAdmin
2. db.createUser( { user: "accountJTP",
3. pwd: "**<cleartext** password**>**",
4. customData: { Employee: 12345 },
5. roles: [ { role: "clusterAdmin", db: "admin" },
6. { role: "readAnyDatabase", db: "admin" },
7. "readWrite"] },
8. w: "majority" , wtimeout: 5000 } )

**Output:**



#4. db.dropUser(<user>, <writeConcern>)

The db.dropUser() method wraps the dropUser command and removes the user from the current database before dropping a user who has the userAdmin AnyDatabase role. You have to make it clear that you have at least an additional user with user administration privileges.

**Example:**

The following operation drops the jtpAdmin user on the studnet database using db.dropUser().

1. use testwriter
2. db.dropUser("testwriter", {w: "majority", wtimeout: 4000})



#5. db.removeUser(<username>)

There is nothing more use of this method. You can use this method to removes the specified username from the current database.

#6. db.updateUser(<username>, <update>, <writeConcern>)

The updateUser method is used to updates the user's profile of the specified database. Using this method will completely replaces the old field's values. This method adds updates to the user's roles array.

**Syntax:**

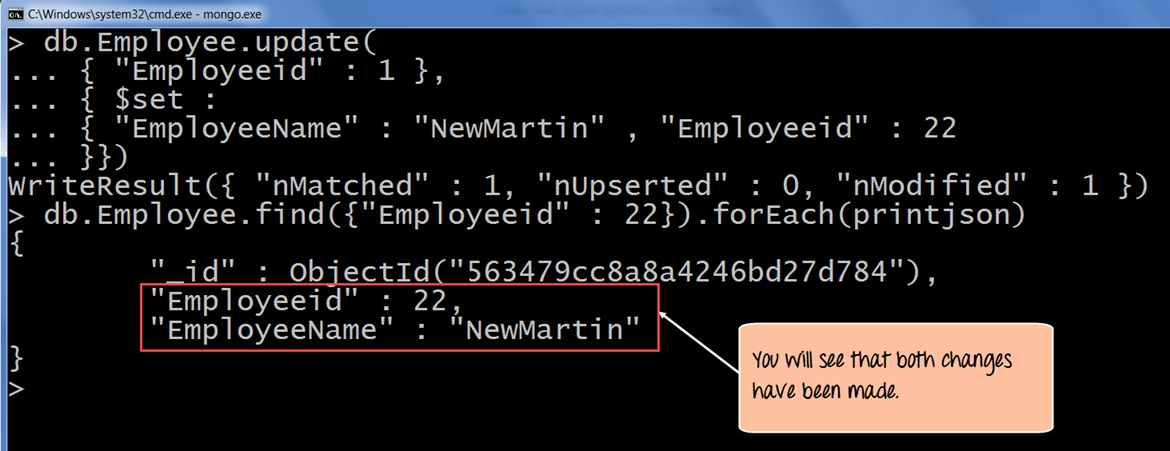
1. db.updateUser(
2. "**<username>**",
3. {
4. customData : { **<any** info.**>** },
5. roles : [
6. { role: "**<role>**", db: "**<database>**" } | "**<role>**",
7. ...
8. ],
9. pwd: "**<cleartext** password**>**",
10. authenticationRestrictions: [
11. {
12. clientSource: ["**<IP>**" | "**<CIDR** range**>**", ...],
13. serverAddress: ["**<IP>**", | "**<CIDR** range**>**", ...]
14. },
15. ...
16. ],
17. mechanisms: [ "**<SCRAM-SHA-1**|SCRAM-SHA-256**>**", ... ],
18. passwordDigestor: "**<server**|client**>**"
19. },
20. writeConcern: { **<write** concern**>** }
21. )

**Example:**

The following example will completely replaces the user's customData and roles data using db.updateUser() method:

1. use Employee
2. db.updateUser( "NewMartin",
3. {
4. customData : { employeeId : "001" },
5. roles : [
6. { role : "read", db : "assets"  }
7. ]
8. } )

**Output:**



MongoDB Replication Methods

The MongoDB Replication methods are used to replicate the member to the replica sets.

#1. rs.add(host, arbiterOnly)

The add method adds a member to the specified replica set. We are required to connect to the primary set of the replica set to this method. The connection to the shell will be terminated if the method will trigger an election for primary. For example - if we try to add a new member with a higher priority than the primary. An error will be reflected by the mongo shell even if the operation succeeds.

**Example:**

In the following example we will add a new secondary member with default vote.

Play Video

Play

Unmute

Current TimeÂ 0:18

/

DurationÂ 4:57

Loaded: 100.00%

Â

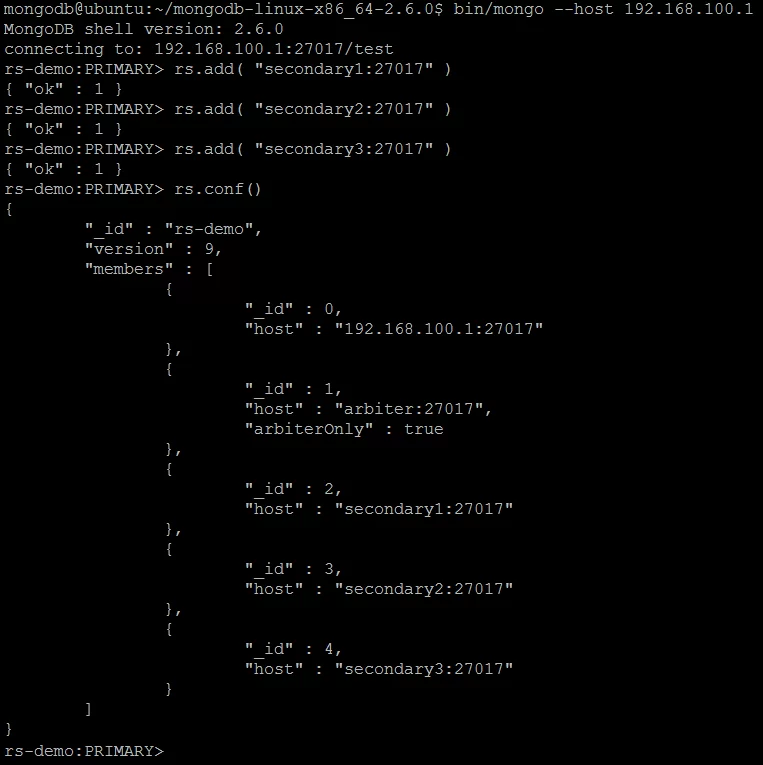
Fullscreen

[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. rs.add( { host: "mongodbd4.example.net:27017" } )

*Host name*

1. rs.add( "mongodbd4.example.net:27017" )



#2. rs.addArb(host)

We can add a new arbiter to an existing replica set at the specified host.

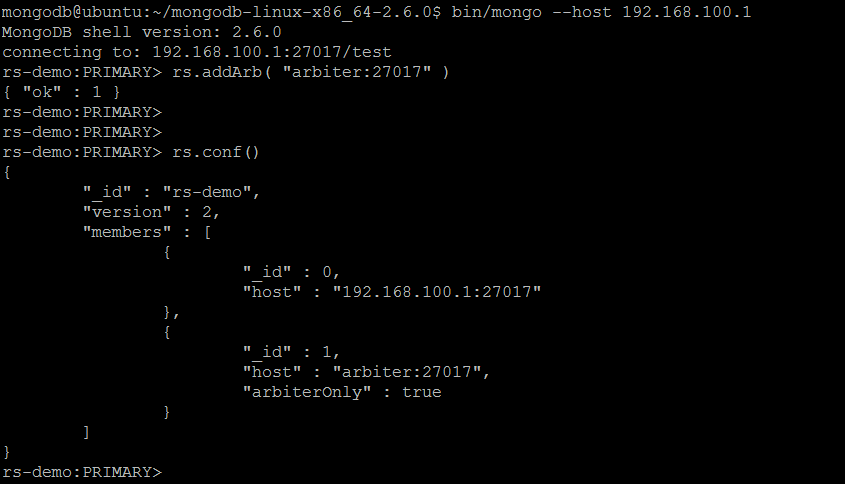
**Example**

Adding an arbiter to the Associated-Hostname

1. mongod --bind\_ip localhost,My-Example-**<Associated-Hostname>**

If you want to connect to this instance, the hostname or its associated IP address 198.52.100.2 must be specified by the remote clients:

1. mongo --host My-Example-**<Associated-Hostname>**
2. mongo --host 198.52.100.2



#3. rs.conf()

The conf method is used to get a document that contains the configuration of the current replica set.

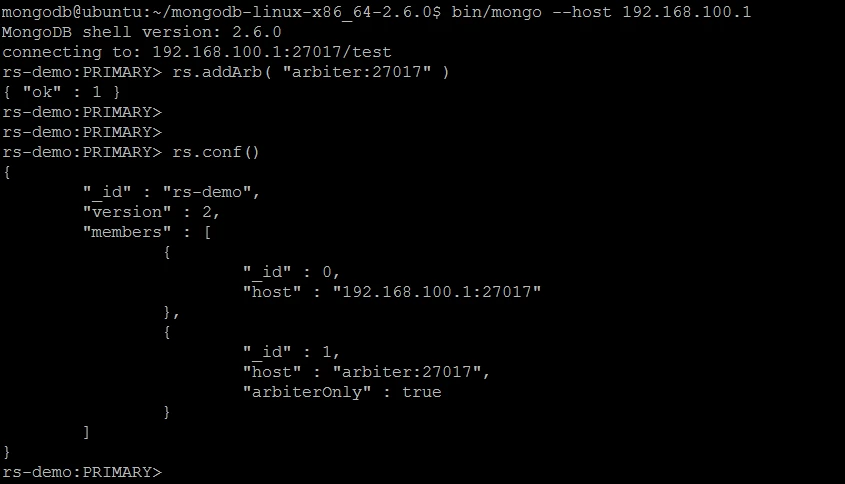
The rs.conf() wraps the replSetGetConfig command configuration.

**Example**

The following script shows the configuration document of a replica set that include the subset of these settings:

1. {
2. \_id: **<string>**,
3. version: **<int>**,
4. protocolVersion: **<number>**,
5. writeConcernMajorityJournalDefault: **<boolean>**,
6. configsvr: **<boolean>**,
7. members: [
8. {
9. \_id: **<int>**,
10. host: **<string>**,
11. arbiterOnly: **<boolean>**,
12. buildIndexes: **<boolean>**,
13. hidden: **<boolean>**,
14. priority: **<number>**,
15. tags: **<document>**,
16. slaveDelay: **<int>**,
17. votes: **<number>**
18. }, ], settings: {
19. chainingAllowed : **<boolean>**,
20. heartbeatIntervalMillis : **<int>**,
21. heartbeatTimeoutSecs: **<int>**,
22. electionTimeoutMillis : **<int>**,
23. catchUpTimeoutMillis : **<int>**,
24. getLastErrorModes : **<document>**,
25. getLastErrorDefaults : **<document>**,
26. replicaSetId: **<ObjectId>**
27. }
28. }

**Output:**



#4. rs.initiate(configuration)

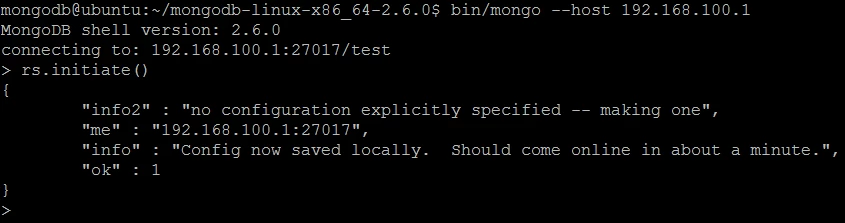
The method initiates a replica set. A document that holds the rs.initiate() configuration of a replica set can be considered by this method, but that is optional.

**Example**

The following example initiates a new replica set with three different members.

1. {
2. \_id: "JTPReplSet",
3. version: 1,
4. members: [
5. { \_id: 0, host : "mongodb0.example.net:27017" },
6. { \_id: 1, host : "mongodb1.example.net:27017" },
7. { \_id: 2, host : "mongodb2.example.net:27017" }
8. ]
9. }

**Output:**



#5. rs.reconfig(configuration, force)

As the name suggests, the method is used to reconfigure an existing replica set. It will overwrite all the existing replica set configuration. We must connect to the primary replica set to run this method.

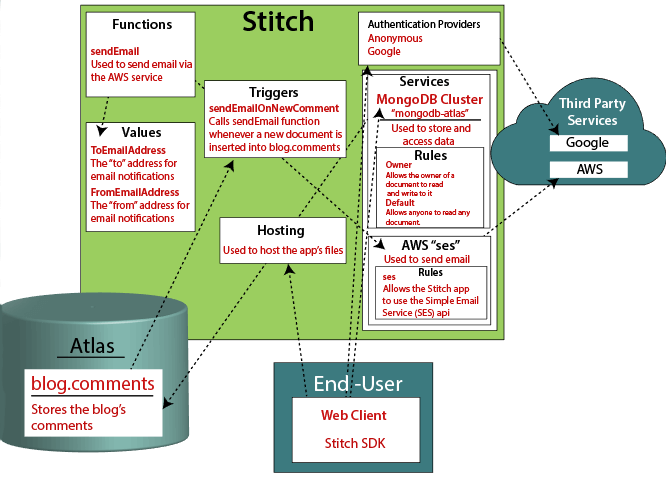
**Examples**

A replica set named rs0 has the following configuration:

1. {
2. "\_id" : "rs0",
3. "version" : 1,
4. "protocolVersion" : NumberLong(1),
5. "members" : [
6. {
7. "\_id" : 0,
8. "host" : "mongodb0.example.net:27017",
9. "arbiterOnly" : false,
10. "buildIndexes" : true,
11. "hidden" : false,
12. "priority" : 1,
13. "tags" : {
15. },
16. "slaveDelay" : NumberLong(0),
17. "votes" : 1
18. },
19. {
20. "\_id" : 1,
21. "host" : "mongodb1.example.net:27017",
22. "arbiterOnly" : false,
23. "buildIndexes" : true,
24. "hidden" : false,
25. "priority" : 1,
26. "tags" : {
28. },
29. "slaveDelay" : NumberLong(0),
30. "votes" : 1
31. },
32. {
33. "\_id" : 2,
34. "host" : "mongodb2.example.net:27017",
35. "arbiterOnly" : false,
36. "buildIndexes" : true,
37. "hidden" : false,
38. "priority" : 1,
39. "tags" : {
41. },
42. "slaveDelay" : NumberLong(0),
43. "votes" : 1
44. }
45. ],
46. "settings" : {
47. "chainingAllowed" : true,
48. "heartbeatIntervalMillis" : 2000,
49. "heartbeatTimeoutSecs" : 10,
50. "electionTimeoutMillis" : 10000,
51. "catchUpTimeoutMillis" : 2000,
52. "getLastErrorModes" : {
54. },
55. "getLastErrorDefaults" : {
56. "w" : 1,
57. "wtimeout" : 0
58. },
59. "replicaSetId" : ObjectId("58858acc1f5609ed986b641b")
60. }
61. }

MongoDB Stitch

MongoDB provides a serverless platform to build an application quickly without setting up server infrastructure. MongoDB Stitch is designed as an upgraded version of MongoDB Atlas. It automatically integrates the connection to our database. Stitch illuminates the development and implementation process. It achieved it by neglecting the requirement of building and deploying our backend. MongoDB stitch is available as a backend service that allows us to configure data-authentication, data access rules, and services easily.



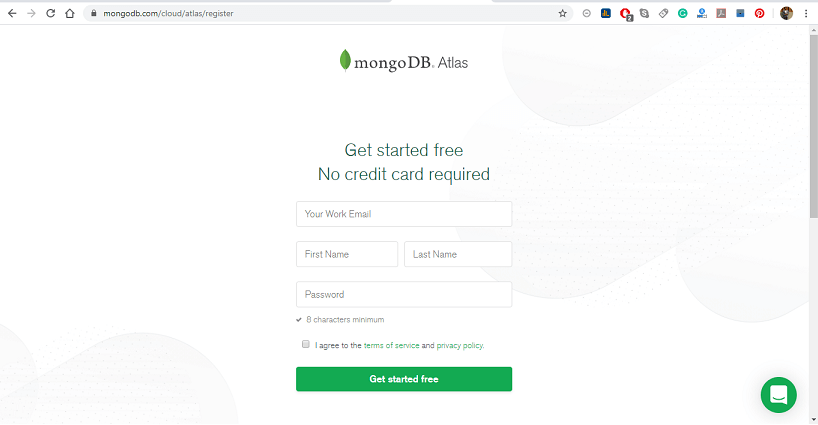
MongoDB stitch provides an upgradable infrastructure design to handle the request. It also coordinates the services and database interactions. i.e., we don't need to spend time and resources on tasks such as configuring our servers, etc.

For example- we can use MongoDB Stitch to generate a pipeline that would allow us to accept payment using Stripe via the HTTP service, update the date of purchase, and send a confirmation email with the Mailgun service.

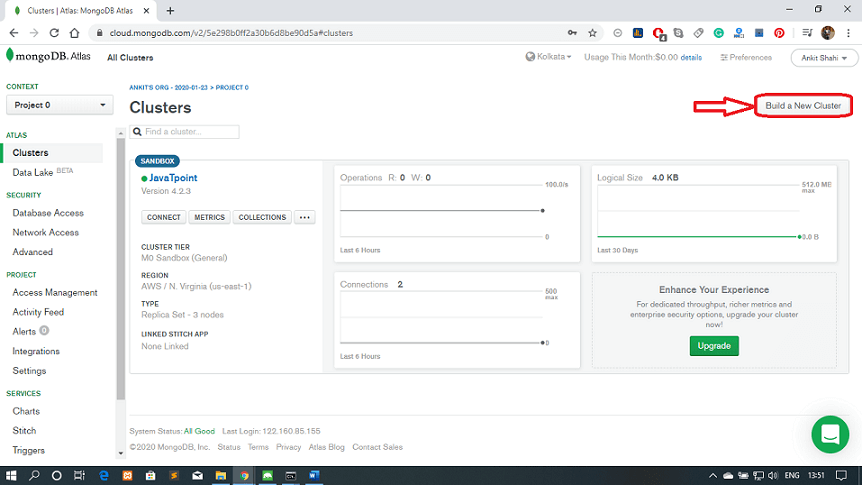
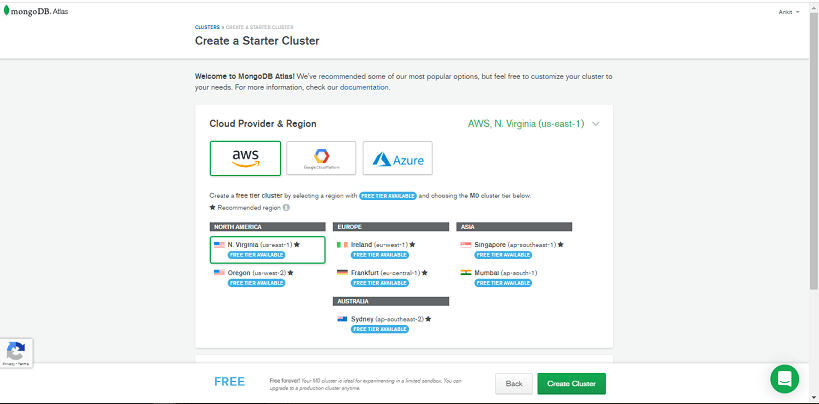
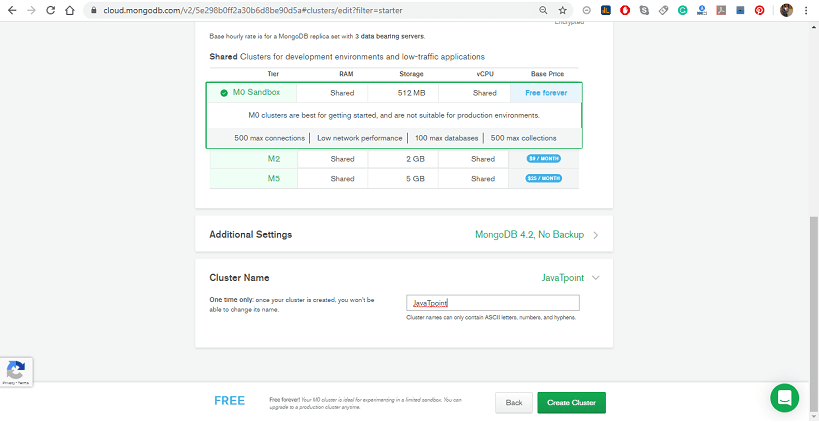
Creating a Stitch App using Stitch UI

**Step 1:** Go to <https://www.mongodb.com/cloud/atlas> page, and log on to your Atlas account.

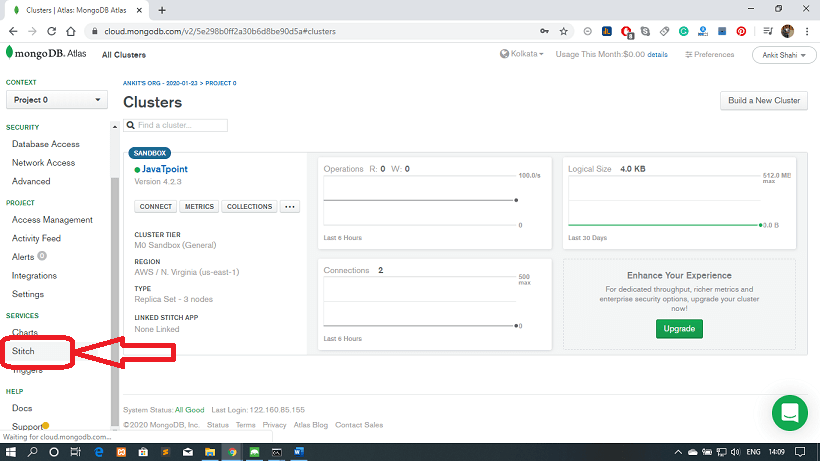
[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)



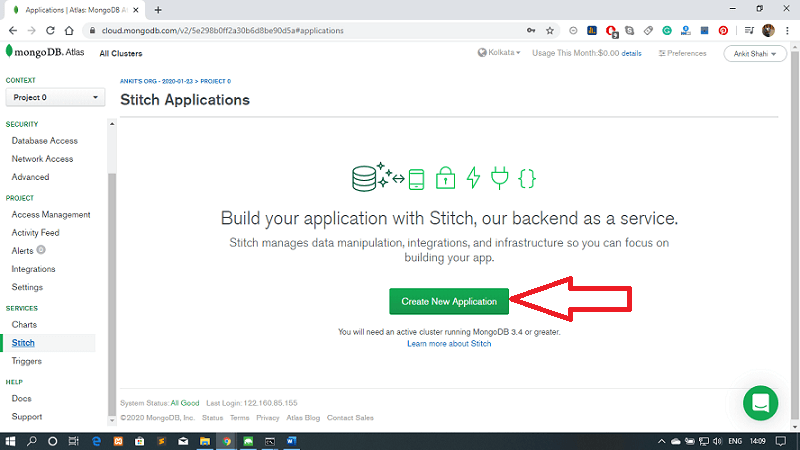
**Step 2:** Now, you have to create a cluster to use with your MongoDB Stitch app, follow these steps:

* Click on the ***Clusters*** button on the left navigation window, then click on the ***Build New Cluster*** button.  
  
* Select your preferred service provider, region, tier, and other required settings.  
  
* The name of the cluster is ***Cluster0*** by default. If you want to change the name of the cluster, you have to do it at this step because the cluster name cannot be changed once configured.  
  
* Finally, click the Create Cluster button to save the changes you made.

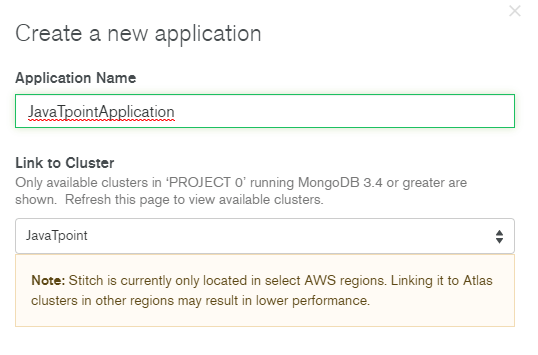
**Step 3:** Inside the [MongoDB Atlas](https://www.javatpoint.com/mongodb-atlas), click ***Stitch Apps*** from the left-hand navigation window.



**Step 4:** After that, click on the ***Create New Application*** button.

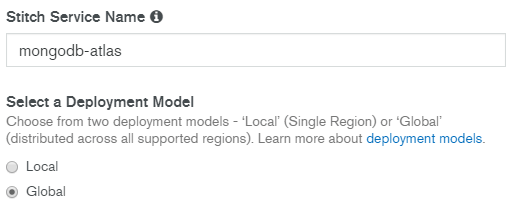


**Step 5:** On the Create a new application pop-up window, enter a Name for your Stitch app.

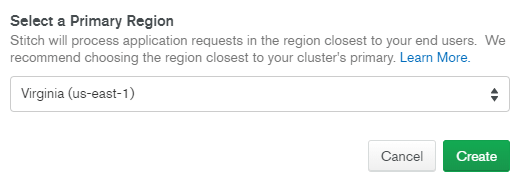


**Step 6:** Choose a cluster in your project from the ***Link to Cluster*** dropdown dialogue. MongoDB Stitch automatically creates a [MongoDB](https://www.javatpoint.com/mongodb-tutorial) service that Is linked to your cluster.

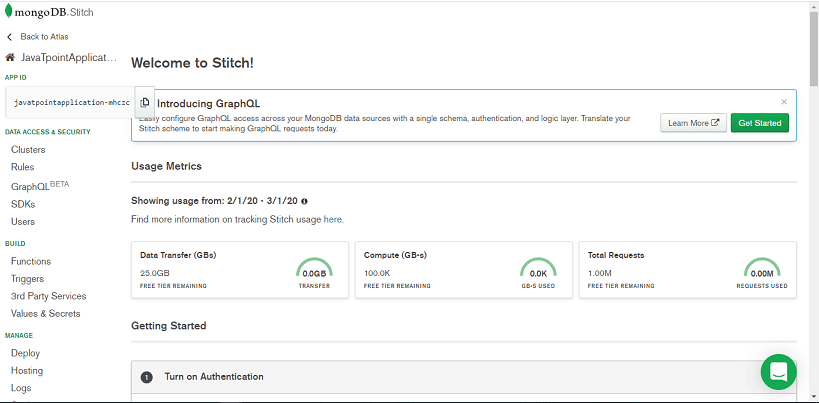
**Step 7:** Fill in the name for the service that Stitch will create in the ***Stitch Service Name*** field.



**Step 8:** Choose a deployment model and deployment region for your application. Then click on the ***Create*** button.



The following window will appear after the successful deployment.



Creating a Stitch App using Stitch CLI

**Step 1:** Firstly, create a new directory for your application and add a stitch.json file to the root level of the directory. The file must contain an empty [JSON object](https://www.javatpoint.com/json-object).

**Step 2:** Authenticate the MongoDB Stitch application with Atlas using an [API](https://www.javatpoint.com/api-full-form) key.

1. stitch-cli login --api-key=my-api-key --private-api-key=my-private-api-key

**Step 3:** Now, import the Stitch CLI library using the "stitch-cli import" command.

**Step 4:** You should verify if your application is created or not.

Query Anywhere with Stitch

Using the MongoDB query language, we can query our stored data in MongoDB directly from our client application code. The Stitch server for the MongoDB collection allows us to securely filter the result using the specified data access rules based on the logged-in user or the contents of each document.

The Student collection contains documents describing each student in an example collage. Each document includes the student's name, email, address, fees, and information on the student's stream. In the example given below, we compare the student's collection for all the documents and return the formatted results as a table.

**HTML File:**

1. <!-- Base Stitch Browser SDK --> **<script** src="https://s3.amazonaws.com/stitch                      sdks/js/bundles/4.0.13/stitch.js"**></script>**
3. **<div** class="results-bar"**>**
4. **<p>**Count of Results:**</p>**
5. **<span** id="num-results" class="results-bar\_\_count"**></span>**
6. **</div>**
7. **<table** class="table table-striped"**>**
8. **<thead** class="thead"**>**
9. **<tr>**
10. **<th>**Name**</th>**
11. **<th>**Email**</th>**
12. **<th>**Class**</th>**
13. **<th>**Batch**</th>**
14. **<th>**Fees**</th>**
15. **</tr>**
16. **</thead>**
17. **<tbody** id='students'**></tbody>**
18. **</table>**

**Java Script File:**

1. **const** {
2. Stitch,
3. RemoteMongoClient,
4. UserPasswordCredential
5. } = stitch;
7. **const** stitchClient = Stitch.initializeDefaultAppClient("stitch-quickstarts-zhpox");
9. login("javatpoint@example.com", "password123").then(() => {
10. // Initialize a MongoDB Service Client
11. **const** mongodb = stitchClient.getServiceClient( RemoteMongoClient.factory,
12. "mongodb-atlas");
13. // Get a hook to the student collection
14. **const** students = mongodb.db("Admin").collection("students");
16. **return** students.find({}, {
17. // limit: 2,
18. // sort: { "fees": -1 }
19. })
20. .asArray();
21. })
22. .then(displayStudents)
24. function login(email, password) {
25. **const** credential = **new** UserPasswordCredential(email, password);
26. **return** stitchClient.auth.loginWithCredential(credential);
27. }

30. // Renders the the students' information in the table
31. function displayStudents(students) {
32. **const** employeesTableBody = document.getElementById("students");
33. **const** numResultsEl = document.getElementById("num-results");
34. **const** tableRows = students.map(student => {
35. **return** '
36. <tr>
37. <td>${student.name.last}, ${student.name.first}</td>
38. <td>${student.email}</td>
39. <td>${student.role}</td>
40. <td>${student.manager.name.first}${student.manager.name.last} (${student.admin.id || "no manager"})</td>
41. <td>${student.fees}</td>
42. </tr>
43. ';
44. });
45. studentTableBody.innerHTML = tableRows.join("");
46. numResultsEl.innerHTML = student.length;
47. }

**Protecting Data with Rules**

We can use collection rules if we don't want to allow every student to see the data of every other student. We can use it to control the data that all the user can access it without altering the pattern of the query.

Creating a Blog App

Here we are creating a blog and commenting system using Stitch. We are using MongoDB Stitch [JavaScript](https://www.javatpoint.com/javascript-tutorial) SDK and the MongoDB service to add and query comments directly from the client code.

**The architecture of the Blog App**

The blog app architecture requires the following features:

* Log in ability.
* Blog post storage ability.
* Blog posts comment ability.

When we use MongoDB Atlas Cluster, we can store comments and the authentication details that allow users to post comments using a temporary account.

The three main components of blog architecture are:

* A web front-end,
* Stitch application,
* MongoDB Atlas database.

The front-end of the blog application architecture handles the display and user interaction. Stitch manages all the request from the front-end and allow only verified requests to the database, which saves comments for our users.

Creating Backend for the Blog App

The backend of the blog app is used to store comments and other details like - authenticate and authorize users, finding existing comments for a blog post etc. We are storing the comments within a sample of MongoDB application. Here, we are going to limit the permission for the user to create, edit, and delete comments only associated with their user ID by authorization. We also need to make sure that a user can't log in as another user; we can achieve this by using the built-in user management system in the MongoDB stitch.

**Requirements:**

* MongoDB Atlas account.
* And a MongoDB cluster that is hosted on Atlas. We suggest you to create an M0 Atlas cluster that is free of cost and good for learning purpose.

**Step 1:** Create a Stitch Application as directed above.

**Step 2:** Turn on the Anonymous Authentication inside the Stitch application that you have created.

**Step 3:** Finally, configure the ***blog.comments*** MongoDB Collection

* Click on rules under MongoDB Atlas from the left-hand navigation of the Stitch UI.
* Then, click on Add Collection.
* Now, enter the blog for the Database Name, and enter comments for the Collection name.
* Select the "No Template" option and click on Add Collection.

**Step 4:** Enable reading and writing to the comments.

**Step 5:** Finally, post your application by clicking on **Review & Deploy Changes** in the pop-up at the top of the Stitch [GUI](https://www.javatpoint.com/gui-full-form).

Creating a Web Client for the blog app

**Step 1:** Create an [HTML](https://www.javatpoint.com/html-tutorial) page as follows.

1. **<html>**
2. **<head>**
3. **</head>**
4. **<body>**
5. **<h3>**This is the first blog post of JavaTpoint**</h3>**
6. **<div** id="content"**>**
7. Learn technology from javaTpoint to keep yourself industry ready.
8. **</div>**
9. **<hr>**
10. **<div** id="comments"**></div>**
11. **</body>**
12. **</html>**

**Step 2:** Now, attach the following JavaScript SDK. To attach the MongoDB Stitch SDK. Add the script tag given below to the head section of your html file.

1. **<script** src="https://s3.amazonaws.com/stitch-sdks/js/bundles/4/stitch.js"**></script>**

**Step 3:** Initialize the app client and a MongoDB Service client to store comments in MongoDB. Replace the app id with your Stitch <app-id>. You find your App ID on the client page of Stitch Interface.

1. **<script>**
2. // it initializing the App Client
3. const client = stitch.Stitch.initializeDefaultAppClient("**<give-your-app-id-here>**");
4. // Now, it will get a MongoDB Service Client
5. const mongodb = client.getServiceClient(
6. stitch.RemoteMongoClient.factory,
7. "mongodb-atlas"
8. );
9. // Getting a reference to the blog database
10. const db = mongodb.db("blog");
11. **</script>**

**Step 4:** Now, add the script given below to query and display the comments when page loads.

1. function displayComments() {
2. db.collection("comments")
3. .find({}, {limit: 1000})
4. .toArray()
5. .then(docs =**>** {
6. const html = docs.map(doc =**>** '**<div>**${doc.comment}**</div>**');
7. document.getElementById("comments").innerHTML = html;
8. });
9. }

**Step 5:** You have to create a file that allows the user to login and display the comment during the loading time.

1. function displayCommentsOnLoad() {
2. client.auth
4. .loginWithCredential(new stitch.AnonymousCredential())
6. .then(displayComments)
7. .catch(console.error);
8. }

**Step 6:** Now, create a form to submit the comments.

1. function addComment() {
2. const newComment = document.getElementById("new\_comment");
3. console.log("add comment", client.auth.user.id)
4. db.collection("firstcomment")
5. .insertOne({ owner\_id : client.auth.user.id, comment: newComment.value })
6. .then(displayComments);
7. newComment.value = "";
8. }

MongoDB Interview Questions

1) What do you understand by NoSQL databases? Is MongoDB a NoSQL database? explain.

At the present time, the internet is loaded with big data, big users, big complexity etc. and also becoming more complex day by day. NoSQL is answer of all these problems, It is not a traditional database management system, not even a relational database management system (RDBMS). NoSQL stands for "Not Only SQL". NoSQL is a type of database that can handle and sort all type of unstructured, messy and complicated data. It is just a new way to think about the database.

Yes. MongoDB is a NoSQL database.

2) Which are the different languages supported by MongoDB?

MonggoDB provides official driver support for C, C++, C#, Java, Node.js, Perl, PHP, Python, Ruby, Scala, Go and Erlang.

You can use MongoDB with any of the above languages. There are some other community supported drivers too but the above mentioned ones are officially provided by MongoDB.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

3) What are the different types of NoSQL databases? Give some example.

NoSQL database can be classified as 4 basic types:

1. Key value store NoSQL database
2. Document store NoSQL database
3. Column store NoSQL database
4. Graph base NoSQL databse

There are many NoSQL databases. MongoDB, Cassandra, CouchBD, Hypertable, Redis, Riak, Neo4j, HBASE, Couchbase, MemcacheDB, Voldemort, RevenDB etc. are the examples of NoSQL databases.

4) Is MongoDB better than other SQL databases? If yes then how?

MongoDB is better than other SQL databases because it allows a highly flexible and scalable document structure.

**For example:**

* One data document in MongoDB can have five columns and the other one in the same collection can have ten columns.
* MongoDB database are faster than SQL databases due to efficient indexing and storage techniques.

5) What type of DBMS is MongoDB?

MongoDB is a document oriented DBMS

6) What is the difference between MongoDB and MySQL?

Although MongoDB and MySQL both are free and open source databases, there is a lot of difference between them in the term of data representation, relationship, transaction, querying data, schema design and definition, performance speed, normalization and many more. To compare MySQL with MongoDB is like a comparison between Relational and Non-relational databases.

7) Why MongoDB is known as best NoSQL database?

MongoDb is the best NoSQL database because, it is:

Document Oriented

Rich Query language

High Performance

Highly Available

Easily Scalable

8) Does MongoDB support primary-key, foreign-key relationship?

No. By Default, MongoDB doesn't support primary key-foreign key relationship.

9) Can you achieve primary key - foreign key relationships in MongoDB?

We can achieve primary key-foreign key relationship by embedding one document inside another. For example: An address document can be embedded inside customer document.

10) Does MongoDB need a lot of RAM?

No. There is no need a lot of RAM to run MongoDB. It can be run even on a small amount of RAM because it dynamically allocates and de-allocates RAM according to the requirement of the processes.

11) Explain the structure of ObjectID in MongoDB.

ObjectID is a 12-byte BSON type. These are:

* 4 bytes value representing seconds
* 3 byte machine identifier
* 2 byte process id
* 3 byte counter

12) Is it true that MongoDB uses BSON to represent document structure?

Yes.

13) What are Indexes in MongoDB?

In MondoDB, Indexes are used to execute query efficiently. Without indexes, MongoDB must perform a collection scan, i.e. scan every document in a collection, to select those documents that match the query statement. If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.

14) By default, which index is created by MongoDB for every collection?

By default, the\_id collection is created for every collection by MongoDB.

15) What is a Namespace in MongoDB?

Namespace is a concatenation of the database name and the collection name. Collection, in which MongoDB stores BSON objects.

16) Can journaling features be used to perform safe hot backups?

Yes.

17) Why does Profiler use in MongoDB?

MongoDB uses a database profiler to perform characteristics of each operation against the database. You can use a profiler to find queries and write operations

18) If you remove an object attribute, is it deleted from the database?

Yes, it be. Remove the attribute and then re-save() the object.

19) In which language MongoDB is written?

MongoDB is written and implemented in C++.

20) Does MongoDB need a lot space of Random Access Memory (RAM)?

No. MongoDB can be run on small free space of RAM.

21) What language you can use with MongoDB?

MongoDB client drivers supports all the popular programming languages so there is no issue of language, you can use any language that you want.

22) Does MongoDB database have tables for storing records?

No. Instead of tables, MongoDB uses "Collections" to store data.

23) Do the MongoDB databases have schema?

Yes. MongoDB databases have dynamic schema. There is no need to define the structure to create collections.

24) What is the method to configure the cache size in MongoDB?

MongoDB's cache is not configurable. Actually MongoDb uses all the free spaces on the system automatically by way of memory mapped files.

25) How to do Transaction/locking in MongoDB?

MongoDB doesn't use traditional locking or complex transaction with Rollback. MongoDB is designed to be light weighted, fast and predictable to its performance. It keeps transaction support simple to enhance performance.

26) Why 32 bit version of MongoDB are not preferred ?

Because MongoDB uses memory mapped files so when you run a 32-bit build of MongoDB, the total storage size of server is 2 GB. But when you run a 64-bit build of MongoDB, this provides virtually unlimited storage size. So 64-bit is preferred over 32-bit.

27) Is it possible to remove old files in the moveChunk directory?

Yes, These files can be deleted once the operations are done because these files are made as backups during normal shard balancing operation. This is a manual cleanup process and necessary to free up space.

28) What will have to do if a shard is down or slow and you do a query?

If a shard is down and you even do query then your query will be returned with an error unless you set a partial query option. But if a shard is slow them Mongos will wait for them till response.

29)Explain the covered query in MongoDB.

A query is called covered query if satisfies the following two conditions:

* The fields used in the query are part of an index used in the query.
* The fields returned in the results are in the same index.

30) What is the importance of covered query?

Covered query makes the execution of the query faster because indexes are stored in RAM or sequentially located on disk. It makes the execution of the query faster.

Covered query makes the fields are covered in the index itself, MongoDB can match the query condition as well as return the result fields using the same index without looking inside the documents.

31) What is sharding in MongoDB?

In MongoDB, Sharding is a procedure of storing data records across multiple machines. It is a MongoDB approach to meet the demands of data growth. It creates horizontal partition of data in a database or search engine. Each partition is referred as shard or database shard.

32) What is replica set in MongoDB?

A replica can be specified as a group of mongo instances that host the same data set. In a replica set, one node is primary, and another is secondary. All data is replicated from primary to secondary nodes.

33) What is primary and secondary replica set in MongoDB?

In MongoDB, primary nodes are the node that can accept write. These are also known as master nodes. The replication in MongoDB is single master so, only one node can accept write operations at a time.

Secondary nodes are known as slave nodes. These are read only nodes that replicate from the primary.

34) By default, which replica sets are used to write data?

By default, MongoDB writes data only to the primary replica set.

35) What is CRUD in MongoDB?

MongoDB supports following CRUD operations:

* Create
* Read
* Update
* Delete

36) In which format MongoDB represents document structure?

MongoDB uses BSON to represent document structures.

37) What will happen when you remove a document from database in MongoDB? Does MongoDB remove it from disk?

Yes. If you remove a document from database, MongoDB will remove it from disk too.

38) Why are MongoDB data files large in size?

MongoDB doesn't follow file system fragmentation and pre allocates data files to reserve space while setting up the server. That's why MongoDB data files are large in size.

39) What is a storage engine in MongoDB?

A storage engine is the part of a database that is used to manage how data is stored on disk.

**For example:** one storage engine might offer better performance for read-heavy workloads, and another might support a higher-throughput for write operations.

40) Which are the storage engines used by MongoDB?

MMAPv1 and WiredTiger are two storage engine used by MongoDB.

41) What is the usage of profiler in MongoDB?

A database profiler is used to collect data about MongoDB write operations, cursors, database commands on a running mongod instance. You can enable profiling on a per-database or per-instance basis.

The database profiler writes all the data it collects to the system. profile collection, which is a capped collection.

42) Is it possible to configure the cache size for MMAPv1 in MongoDB?

No. it is not possible to configure the cache size for MMAPv1 because MMAPv1 does not allow configuring the cache size.

43) How to configure the cache size for WiredTiger in MongoDB?

For the WiredTiger storage engine, you can specify the maximum size of the cache that WiredTiger will use for all data. This can be done using storage.wiredTiger.engineConfig.cacheSizeGB option.

44) How does MongoDB provide concurrency?

MongoDB uses reader-writer locks for concurrency. Reader-writer locks allow concurrent readers shared access to a resource, such as a database or collection, but give exclusive access to a single write operation.

45) What is the difference between MongoDB and Redis database?

**Difference between MongoDB and Redis:**

* Redis is faster than MongoDB.
* Redis has a key-value storage whereas MongoDB has a document type storage.
* Redis is hard to code but MongoDB is easy.

For more information: [click here](https://www.javatpoint.com/redis-vs-mongodb)

46) What is the difference between MongoDB and CouchDB?

**Difference between MongoDB and CouchDB:**

* MongoDB is faster than CouchDB while CouchDB is safer than MongoDB.
* Triggers are not available in MongoDB while triggers are available in CouchDB.
* MongoDB serializes JSON data to BSON while CouchDB doesn't store data in JSON format.

For more information: [click here](https://www.javatpoint.com/couchdb-vs-mongodb)

47) What is the difference between MongoDB and Cassandra?

**Difference between MongoDB and Cassandra:**

* MongoDB is cross-platform document-oriented database system while Cassandra is high performance distributed database system.
* MongoDB is written in C++ while Cassandra is written in Java.
* MongoDB is easy to administer in the case of failure while Cassandra provides high availability with no single point of failure.

For more information: [click here](https://www.javatpoint.com/cassandra-vs-mongodb)

48) Is there any need to create database command in MongoDB?

You don't need to create a database manually in MongoDB because it creates automaically when you save the value into the defined collection at first time.

# MongoDB Create Database

**Use Database method:**

There is no create database command in MongoDB. Actually, MongoDB do not provide any command to create database.

It may be look like a weird concept, if you are from traditional SQL background where you need to create a database, table and insert values in the table manually.

Here, in MongoDB you don't need to create a database manually because MongoDB will create it automatically when you save the value into the defined collection at first time.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

You also don't need to mention what you want to create, it will be automatically created at the time you save the value into the defined collection.

#### **Here one thing is very remarkable that you can create collection manually by "db.createCollection()" but not the database.**

## How and when to create database

If there is no existing database, the following command is used to create a new database.

**Syntax:**

1. use DATABASE\_NAME

If the database already exists, it will return the existing database.

Let' take an example to demonstrate how a database is created in [MongoDB](https://www.javatpoint.com/mongodb-tutorial). In the following example, we are going to create a database "javatpointdb".

**See this example**

1. >use javatpointdb

Swithched to db javatpointdb

To **check the currently selected database**, use the command db:

1. >db

javatpointdb

To **check the database list**, use the command show dbs:

1. >show dbs

local 0.078GB

Here, your created database "javatpointdb" is not present in the list, **insert at least one document** into it to display database:

1. >db.movie.**insert**({"name":"javatpoint"})

WriteResult({ "nInserted": 1})

1. >show dbs

javatpointdb 0.078GB

local 0.078GB