# **MongoDB**

**Overview**

MongoDB is an [open source](https://whatis.techtarget.com/definition/open-source) database management system (DBMS) that uses a document-oriented database model which supports various forms of data. It is one of numerous nonrelational [database](https://searchsqlserver.techtarget.com/definition/database) technologies which arose in the mid-2000s under the [NoSQL](https://searchdatamanagement.techtarget.com/definition/NoSQL-Not-Only-SQL) banner for use in big data applications and other processing jobs involving data that doesn't fit well in a rigid relational model. Instead of using [tables](https://whatis.techtarget.com/definition/table) and [rows](https://searchoracle.techtarget.com/definition/row) as in [relational databases](https://searchdatamanagement.techtarget.com/definition/relational-database), the MongoDB architecture is made up of collections and documents.

MongoDB is a cross-platform, document-oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.

## **Database**

Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

## **Collection**

Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

## **Document**

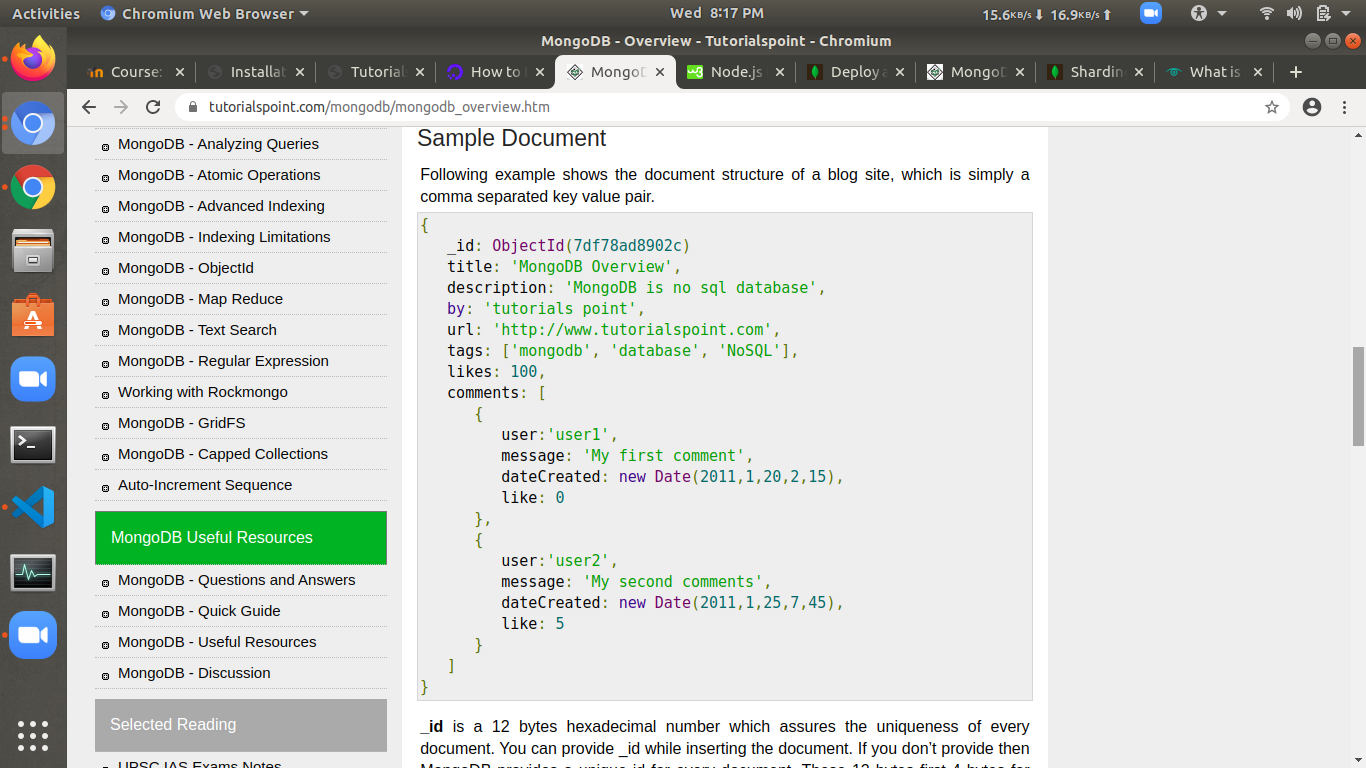
A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

The following table shows the relationship of RDBMS terminology with MongoDB.

|  |  |
| --- | --- |
| **RDBMS** | **MongoDB** |
| Database | Database |
| Table | Collection |
| Tuple/Row | Document |
| column | Field |
| Table Join | Embedded Documents |
| Primary Key | Primary Key (Default key \_id provided by MongoDB itself) |
| **Database Server and Client** |  |
| mysqld/Oracle | mongod |
| mysql/sqlplus | mongo |

## Sample Document

Following example shows the document structure of a blog site, which is simply a comma separated key value pair.



**\_id** is a 12 bytes hexadecimal number which assures the uniqueness of every document. You can provide \_id while inserting the document. If you don’t provide then MongoDB provides a unique id for every document. These 12 bytes first 4 bytes for the current timestamp, next 3 bytes for machine id, next 2 bytes for process id of MongoDB server and remaining 3 bytes are simple incremental VALUE.

## **Advantages of MongoDB over RDBMS**

* **Schema less** − MongoDB is a document database in which one collection holds different documents. Number of fields, content and size of the document can differ from one document to another.
* Structure of a single object is clear.
* No complex joins.
* Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
* Tuning.
* **Ease of scale-out** − MongoDB is easy to scale.
* Conversion/mapping of application objects to database objects not needed.
* Uses internal memory for storing the (windowed) working set, enabling faster access of data.

## **Why Use MongoDB?**

* **Document Oriented Storage** − Data is stored in the form of JSON style documents.
* Index on any attribute
* Replication and high availability
* Auto-Sharding
* Rich queries
* Fast in-place updates
* Professional support by MongoDB

## **Where to Use MongoDB?**

* Big Data
* Content Management and Delivery
* Mobile and Social Infrastructure
* User Data Management
* Data Hub

## **Install MongoDB Driver**

After you’ve created your project with npm init, you can install the MongoDB driver and its dependencies with the command:

**npm install mongodb --save**

This will download the MongoDB driver and add a dependency entry in your package.json file.

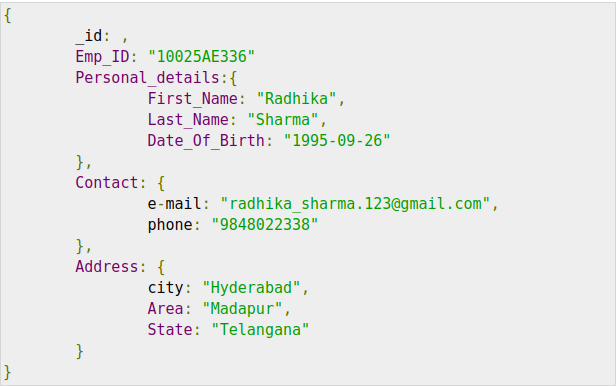
## **Data Model Design**

MongoDB provides two types of data models: — Embedded data model and Normalized data model. Based on the requirement, you can use either of the models while preparing your document.

### **Embedded Data Model**

In this model, you can have (embed) all the related data in a single document, it is also known as de-normalized data model.

For example, assume we are getting the details of employees in three different documents namely, Personal\_details, Contact and, Address, you can embed all the three documents in a single one as shown below −



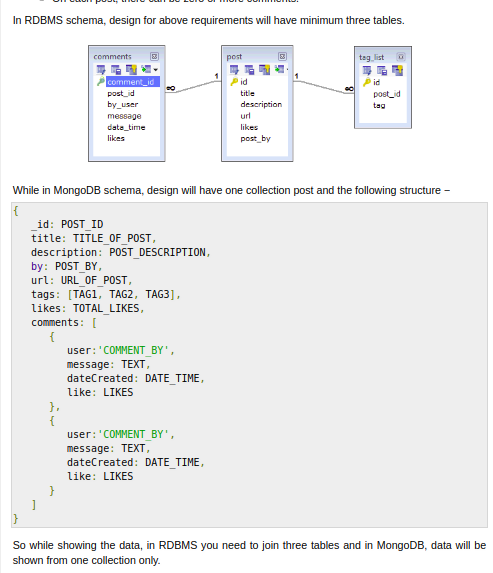
### **Normalized Data Model**

In this model, you can refer the sub documents in the original document, using references. For example, you can re-write the above document in the normalized model as:



## **Considerations while designing Schema in MongoDB**

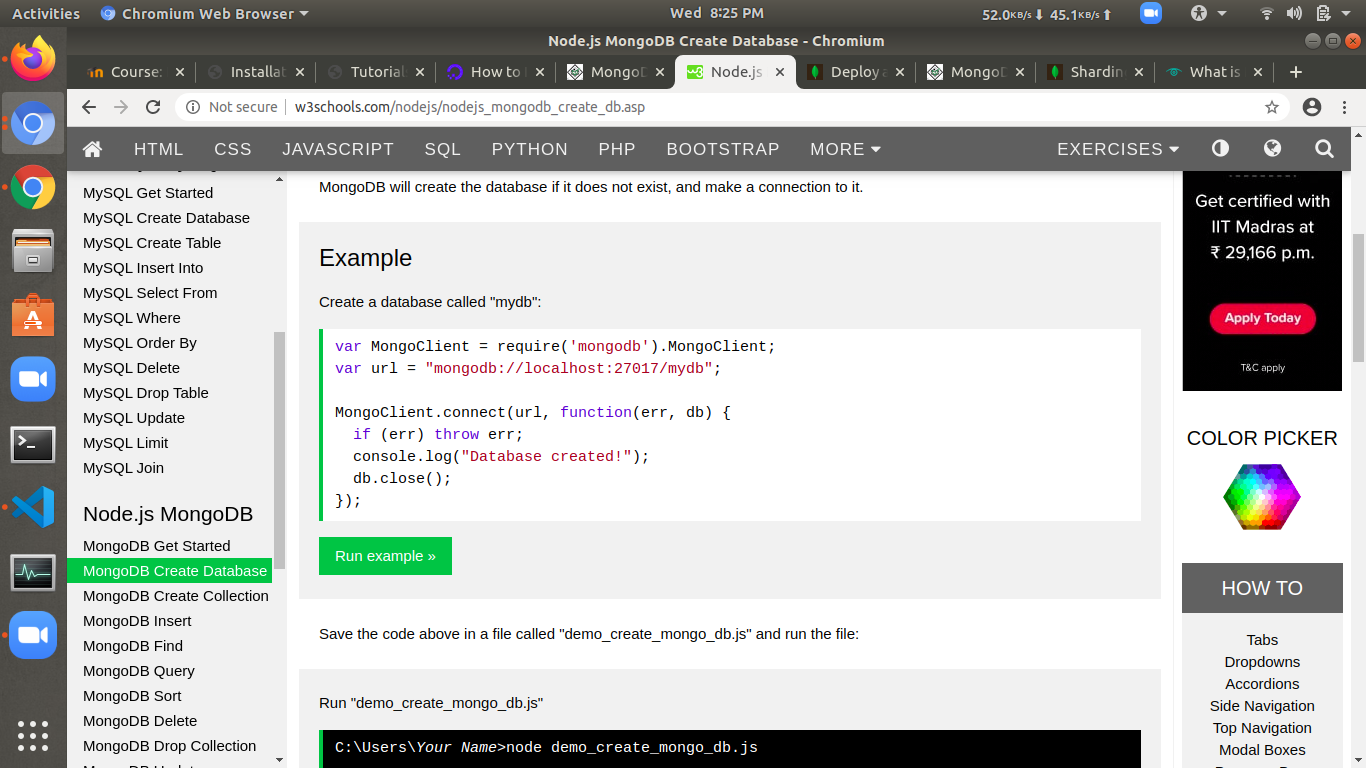
* Design your schema according to user requirements.
* Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
* Duplicate the data (but limited) because disk space is cheap as compare to compute time.
* Do joins while write, not on read.
* Optimize your schema for most frequent use cases.
* Do complex aggregation in the schema.



## **Creating a Database**

To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database you want to create.

MongoDB will create the database if it does not exist and make a connection to it.



Run "demo\_create\_mongo\_db.js"

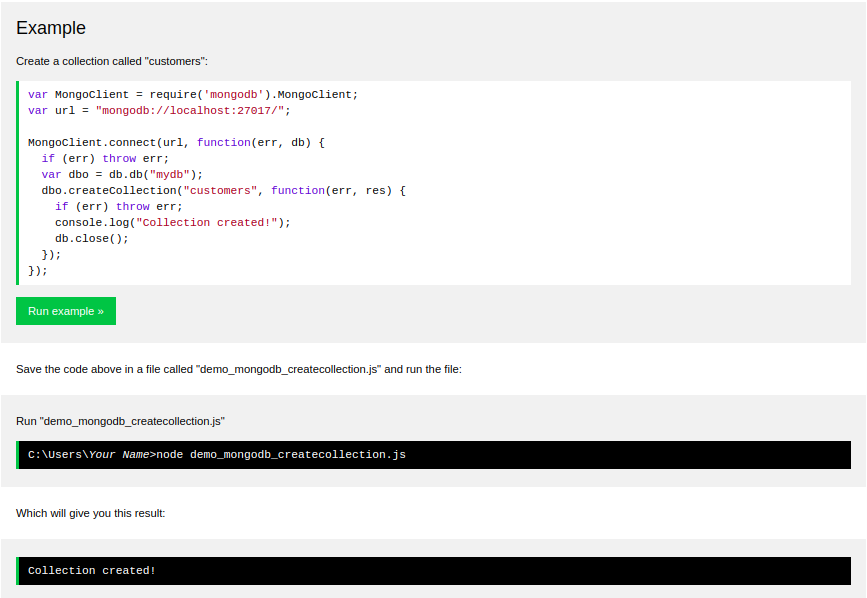
C:\Users\*Your Name*>node demo\_create\_mongo\_db.js

Which will give you this result:

Database created

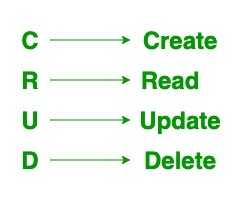
## **Creating a Collection**

To create a collection in MongoDB, use the createCollection() method:



# **MongoDB CRUD operations**

As we know that we can use MongoDB for various things like building an application (including web and mobile), or analysis of data, or an administrator of a MongoDB database, in all these cases we need to interact with the MongoDB server to perform certain operations like entering new data into the application, updating data into the application, deleting data from the application, and reading the data of the application.

MongoDB provides a set of some basic but most essential operations that will help you to easily interact with the MongoDB server and these operations are known as **CRUD operations**.  


## **Create Operations –**

The create or insert operations are used to insert or add new documents in the collection. If a collection does not exist, then it will create a new collection in the database. You can perform, create operations using the following methods provided by the MongoDB:

|  |  |
| --- | --- |
| **METHOD** | **DESCRIPTION** |
| **db.collection.insertOne()** | It is used to insert a single document in the collection. |
| **db.collection.insertMany()** | It is used to insert multiple documents in the collection. |

## **Read Operations –**

The Read operations are used to retrieve documents from the collection, or in other words, read operations are used to query a collection for a document. You can perform read operation using the following method provided by the MongoDB:

|  |  |
| --- | --- |
| **METHOD** | **DESCRIPTION** |
| **db.collection.find()** | It is used to retrieve documents from the collection. |

## **Update Operations –**

The update operations are used to update or modify the existing document in the collection. You can perform update operations using the following methods provided by the MongoDB:

|  |  |
| --- | --- |
| **METHOD** | **DESCRIPTION** |
| **db.collection.updateOne()** | It is used to update a single document in the collection that satisfy the given criteria. |
| **db.collection.updateMany()** | It is used to update multiple documents in the collection that satisfy the given criteria. |
| **db.collection.replaceOne()** | It is used to replace single document in the collection that satisfy the given criteria. |

## **Delete Operations –**

The delete operation are used to delete or remove the documents from a collection. You can perform delete operations using the following methods provided by the MongoDB:

|  |  |
| --- | --- |
| **METHOD** | **DESCRIPTION** |
| **db.collection.deleteOne()** | It is used to delete a single document from the collection that satisfy the given criteria. |
| **db.collection.deleteMany()** | It is used to delete multiple documents from the collection that satisfy the given criteria. |

# **Administration**

The administration documentation addresses the ongoing operation and maintenance of MongoDB instances and deployments. This documentation includes both high level overviews of these concerns as well as tutorials that cover specific procedures and processes for operating MongoDB.

**Administration Concepts**  
The core administration documents address strategies and practices used in the [operation of MongoDB systems](https://intellipaat.com/blog/tutorial/mongodb-tutorial/) and deployments.

* Operational Strategies –Higher level documentation of key concepts for the operation and maintenance of MongoDB deployments.
* MongoDB Backup Methods – Describes approaches and considerations for backing up a MongoDB database.
* Monitoring for MongoDB – An overview of monitoring tools, diagnostic strategies, and approaches to monitoring replica sets and sharded clusters.
* Run-time Database Configuration – Outlines common MongoDB configurations and examples of best-practice configurations for common use cases.
* Data Management – Core documentation that addresses issues in data management, organization, maintenance, and lifecycle management.
* Data Center Awareness – Presents the MongoDB features that allow application developers and [database administrators](https://intellipaat.com/database-architect-training/) to configure their deployments to be more data center aware or allow operational and location-based separation.
* Capped Collections – Capped collections provide a special type of size-constrained collections that preserve insertion order and can support high volume inserts.

Expire Data from Collections by Setting TTL – TTL collections make it possible to automatically remove data from a collection based on the value of a timestamp and are useful for managing data like machine generated event data that are only useful for a limited period of time

**Administration Tutorials**  
The administration tutorials provide specific step-by-step instructions for performing common MongoDB setup, maintenance and configuration operations.

* Configuration, Maintenance, and Analysis – Describes routine management operations, including configuration and performance analysis.
* Manage mongod Processes – Start, configure, and manage running mongod.
* Rotate Log Files – Archive the current log files and start new ones. Continue reading from Configuration, Maintenance, and Analysis for additional tutorials of fundamental MongoDB maintenance procedures.
* Backup and Recovery – Outlines procedures for data backup and restoration with mongod instances and deployments.
* Recover Data after an Unexpected Shutdown – Recover data from MongoDB data files that were not properly closed or have an invalid state.
* MongoDB Scripting – An introduction to the scripting capabilities of the mongo shell and the scripting capabilities embedded in MongoDB instances.

# **Administration Commands**

|  |  |
| --- | --- |
| **Name** | **Description** |
| [clean](https://docs.mongodb.com/manual/reference/command/clean/#dbcmd.clean) | Internal namespace administration command. |
| [cloneCollection](https://docs.mongodb.com/manual/reference/command/cloneCollection/#dbcmd.cloneCollection) | Copies a collection from a remote host to the current host. |
| [cloneCollectionAsCapped](https://docs.mongodb.com/manual/reference/command/cloneCollectionAsCapped/#dbcmd.cloneCollectionAsCapped) | Copies a non-capped collection as a new [capped collection](https://docs.mongodb.com/manual/reference/glossary/#term-capped-collection). |
| [collMod](https://docs.mongodb.com/manual/reference/command/collMod/#dbcmd.collMod) | Add options to a collection or modify a view definition. |
| [compact](https://docs.mongodb.com/manual/reference/command/compact/#dbcmd.compact) | Defragments a collection and rebuilds the indexes. |
| [connPoolSync](https://docs.mongodb.com/manual/reference/command/connPoolSync/#dbcmd.connPoolSync) | Internal command to flush connection pool. |
| [convertToCapped](https://docs.mongodb.com/manual/reference/command/convertToCapped/#dbcmd.convertToCapped) | Converts a non-capped collection to a capped collection. |
| [create](https://docs.mongodb.com/manual/reference/command/create/#dbcmd.create) | Creates a collection or a view. |
| [createIndexes](https://docs.mongodb.com/manual/reference/command/createIndexes/#dbcmd.createIndexes) | Builds one or more indexes for a collection. |
| [currentOp](https://docs.mongodb.com/manual/reference/command/currentOp/#dbcmd.currentOp) | Returns a document that contains information on in-progress operations for the database instance. |
| [drop](https://docs.mongodb.com/manual/reference/command/drop/#dbcmd.drop) | Removes the specified collection from the database. |
| [dropDatabase](https://docs.mongodb.com/manual/reference/command/dropDatabase/#dbcmd.dropDatabase) | Removes the current database. |
| [dropConnections](https://docs.mongodb.com/manual/reference/command/dropConnections/#dbcmd.dropConnections) | Drops outgoing connections to the specified list of hosts. |
| [dropIndexes](https://docs.mongodb.com/manual/reference/command/dropIndexes/#dbcmd.dropIndexes) | Removes indexes from a collection. |
| [filemd5](https://docs.mongodb.com/manual/reference/command/filemd5/#dbcmd.filemd5) | Returns the [md5](https://docs.mongodb.com/manual/reference/glossary/#term-md5) hash for files stored using [GridFS](https://docs.mongodb.com/manual/reference/glossary/#term-gridfs). |
| [fsync](https://docs.mongodb.com/manual/reference/command/fsync/#dbcmd.fsync) | Flushes pending writes to the storage layer and locks the database to allow backups. |
| [fsyncUnlock](https://docs.mongodb.com/manual/reference/command/fsyncUnlock/#dbcmd.fsyncUnlock) | Unlocks one fsync lock. |
| [getParameter](https://docs.mongodb.com/manual/reference/command/getParameter/#dbcmd.getParameter) | Retrieves configuration options. |
| [killCursors](https://docs.mongodb.com/manual/reference/command/killCursors/#dbcmd.killCursors) | Kills the specified cursors for a collection. |
| [killOp](https://docs.mongodb.com/manual/reference/command/killOp/#dbcmd.killOp) | Terminates an operation as specified by the operation ID. |
| [listCollections](https://docs.mongodb.com/manual/reference/command/listCollections/#dbcmd.listCollections) | Returns a list of collections in the current database. |
| [listDatabases](https://docs.mongodb.com/manual/reference/command/listDatabases/#dbcmd.listDatabases) | Returns a document that lists all databases and returns basic database statistics. |
| [listIndexes](https://docs.mongodb.com/manual/reference/command/listIndexes/#dbcmd.listIndexes) | Lists all indexes for a collection. |
| [logRotate](https://docs.mongodb.com/manual/reference/command/logRotate/#dbcmd.logRotate) | Rotates the MongoDB logs to prevent a single file from taking too much space. |
| [reIndex](https://docs.mongodb.com/manual/reference/command/reIndex/#dbcmd.reIndex) | Rebuilds all indexes on a collection. |
| [renameCollection](https://docs.mongodb.com/manual/reference/command/renameCollection/#dbcmd.renameCollection) | Changes the name of an existing collection. |
| [setFeatureCompatibilityVersion](https://docs.mongodb.com/manual/reference/command/setFeatureCompatibilityVersion/#dbcmd.setFeatureCompatibilityVersion) | Enables or disables features that persist data that are backwards-incompatible. |
| [setParameter](https://docs.mongodb.com/manual/reference/command/setParameter/#dbcmd.setParameter) | Modifies configuration options. |
| [shutdown](https://docs.mongodb.com/manual/reference/command/shutdown/#dbcmd.shutdown) | Shuts down the [mongod](https://docs.mongodb.com/manual/reference/program/mongod/#bin.mongod) or [mongos](https://docs.mongodb.com/manual/reference/program/mongos/#bin.mongos) process. |

# **Security**

MongoDB provides various features, such as authentication, access control, encryption, to secure your MongoDB deployments.

### **Authentication**

Authentication is the process of verifying the identity of a client that is trying to connect with a database. MongoDB offers various methods to verify a client's identity. Challenge-based default measures include:

* **SCRAM-SHA-1:** Salted Challenge Authentication Mechanism uses simple text-based usernames and passwords transmitted over a channel protected by transport layer security (TLS).
* **MongoDB-CR**: Like SCRAM, MongoDB-CR verifies a username and password against an authentication database. MongoDB-CR was removed from Version 3.0, and only older iterations use it today.

Either method sends passwords encrypted, and a different hash is generated for each new session so no one can sniff them out.

MongoDB can employ external authentication protocols as well:

* **LDAP**: Lightweight Directory Access Protocol allows users to log in using their centralized passwords. LDAP is designed to help anyone locate and access information they need in either a public or private network.
* **Kerberos**: This is a secret key authentication protocol for server-client interactions. Using Kerberos, users can log in only once using an access ticket.

#### **MongoDB authentication best practices**

First things first: Always enable auth on a production install. For Version 3.5 onwards, this feature is enabled by default.

* Enable access control and use one of MongoDB's authentication mechanisms mentioned above. If your deployment is clustered, each instance should be individually configured.
* Always start by creating an administrator user. Then add additional users as needed.
* Encrypt all communications between [mongod](https://docs.mongodb.com/manual/reference/program/mongod/) and [mongos](https://docs.mongodb.com/manual/reference/program/mongos/) instances as well as internal and external communications using TLS/SSL.
* Encrypt data on each MongoDB host using filesystem, device, or physical encryption.
* Run MongoDB on a trusted network only. Do not allow your database to be routable outbound to the public internet, even when inside a trusted network, and don't let it run on any more interfaces than it has to. This prevents a bad actor from having a means of moving your data from the server to another offsite location (for hardware, that is—software-based routers and static routing tables can still be modified by hackers).
* Make a habit of tracking changes in both the database and data. If you are using MongoDB Enterprise, then use its auditing facility for analysis.

### **Authorization/role-based security**

Role-based access control (RBAC) is one of MongoDB's best features. While you can find well-defined roles within MongoDB that can cover most users, custom roles can be created as well.

A role essentially determines what permissions a user has and what he/she can access. Once a user has been defined by a role, the user cannot access the system beyond it.

You can enable authorization using the **--auth** or the security-authorization setting. **--auth** enables authorization to control a user's access to a database and its resources. This feature also enforces authentication after it's enabled—it requires all clients to verify their identities before being given access.

### **TLS/SSL encryption**

MongoDB offers network encryption and can pass through disk encryption to help you protect your database and communications. TLS and SSL are both standard technologies that are used for encrypting network traffic.

As of MongoDB 2.6, both SSL and TLS are supported by the x.509 certificates, and clients can use the latter to authenticate their identities rather than a username and password. A minimum of 128-bit key length is required by MongoDB for all connections when using TLS/SSL.

While MongoDB can use any valid certificate, self-signed certificates are best avoided as there will be no validation of server identity, even though the communication channel will remain encrypted. In such scenarios, databases may become vulnerable to man-in-the-middle attacks.

#### **How to configure mongod and mongos for TLS/SSL**

To use TLS/SSL in your MongoDB deployment, include the following runtime options with mongod and mongos:

* net.ssl.mode set to **requireSSL**. This setting restricts each server to use only TLS/SSL encrypted connections. You can also specify either the value **allowSSL** or **preferSSL** to set up the use of mixed TLS/SSL modes on a port. See net.ssl.mode for details.
* PEMKeyfile with the .pem file that contains the TLS/SSL certificate and key.

### **Hardening your MongoDB database**

While these steps will help your database survive malicious online activity, going the extra mile hardens your defenses even further. "Hardening" essentially refers to a layer-by-layer method of adding security, where each part of a database is given its own security measures.

MongoDB databases come with their own hardening features.

#### **Configuration hardening with IP binding**

For Version 3.6 onwards, MongoDB binds to localhost by default, while for versions 2.6 to 3.4, only binaries from the official MongoDB RPM and DEB packages will bind to localhost by default.

#### **Network hardening with firewalls and VPNs**

* **Firewalls:** These can help you exert much finer control over network communications. Firewalls limit incoming traffic, particularly from untrusted sources. As ransomware and other kinds of attacks target specific ports, having well-configured firewalls is an ideal defense.  
    
  On Linux systems, administrators can use the iptables interface to access the underlying netfilter firewall. On Windows systems, the netsh command-line interface can be used to access the Windows firewall.
* **Virtual private networks (VPN):** VPNs are ideal for connecting two endpoints over a less-than-secure communication network. Depending on features and how they are configured, VPNs allow for certificate validation and encryption protocols. As VPNs provide a secure tunnel between client and server, they can also be used along with self-generated certificates without needing to worry about man-in-the-middle attacks.

# **Create Indexes**

To create an index on a field or fields, pass an index specification document to the createIndex() method:

{ <**field1**>: <**type1**>, <**field2**>: <**type2**> ... }

## **Create an Ascending Index**

For an ascending index type, specify 1 for <type>.

The following example creates an ascending index key for the dateOfBirth field:

**function** createAscendingIndex(db, callback) { *// Get the users collection* **const** collection = db.collection(**'users'**); *// Create the index* collection.createIndex( { dateOfBirth : 1 }, **function**(err, result) { console.log(result); callback(result); });};

## **Create a Descending Index**

For an descending index type, specify -1 for <type>.

The following example specifies a descending index key on the lastName field:

**function** createDescendingIndex(db, callback) { *// Get the documents collection* **const** collection = db.collection(**'users'**); *// Create the index* collection.createIndex( { lastName : -1 }, **function**(err, result) { console.log(result); callback(result); });};

## **Create a Compound Index**

To specify a compound index, use the compoundIndex method.

The following example specifies a compound index key composed of the lastName field sorted in descending order, followed by the dateOfBirth field sorted in ascending order:

**function** createCompoundIndex(db, callback) { *// Get the documents collection* **const** collection = db.collection(**'users'**); *// Create the index* collection.createIndex( { lastName : -1, dateOfBirth : 1 }, **function**(err, result) { console.log(result); callback(result); });};

## **Create a Text Index**

MongoDB also provides [text](https://docs.mongodb.org/manual/core/index-text/) indexes to support text search of string content. Text indexes can include any field whose value is a string or an array of string elements.

This example specifies a text index key for the comments field:

**function** createTextIndex(db, callback) { *// Get the documents collection* **const** collection = db.collection(**'users'**); *// Create the index* collection.createIndex( { comments : **"text"** }, **function**(err, result) { console.log(result); callback(result); });};

## **Create a Hashed Index**

To specify a [hashed](https://docs.mongodb.org/manual/core/index-hashed/) index key, use the hashed method.

This example specifies a hashed index key for the timestamp field:

**function** createHashedIndex(db, callback) { *// Get the documents collection* **const** collection = db.collection(**'users'**); *// Create the index* collection.createIndex( { timestamp : **"hashed"** }, **function**(err, result) { console.log(result); callback(result); });};

## **Create Geospatial Indexes**

There are also helpers for creating the index keys for the various geospatial indexes supported by mongodb.

### Create a 2dsphere Index

To specify a [2dsphere](https://docs.mongodb.org/manual/core/2dsphere/) index key, use one of the geo2dsphere methods.

This example specifies a 2dsphere index on the location field:

**function** create2dSphereIndex(db, callback) { *// Get the documents collection* **const** collection = db.collection(**'users'**); *// Create the index* collection.createIndex( { location : **"2dsphere"** }, **function**(err, result) { console.log(result); callback(result); });};

### **Create a 2d Index**

To specify a [2d](https://docs.mongodb.org/manual/core/2d/) index key, use the geo2d method.A 2d index is for data stored as points on a two-dimensional plane and is intended for legacy coordinate pairs used in MongoDB 2.2 and earlier.

This example specifies a 2d index on the points field:

**function** create2dIndex(db, callback) { *// Get the documents collection* **const** collection = db.collection(**'users'**); *// Create the index* collection.createIndex( { points : **"2d"** }, **function**(err, result) { console.log(result); callback(result); });};

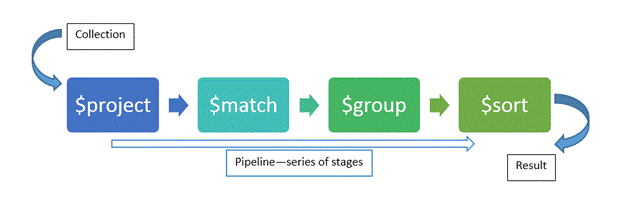
## **IndexOptionn**

In addition to the index specification document, createIndex method can take an index options document, such as to create unique indexes or partial indexes.

# **Aggregation**

## **Overview**

Aggregation operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.



### **Syntax**

Basic syntax of **aggregate()** method is as follows −

>db.COLLECTION\_NAME.aggregate(AGGREGATE\_OPERATION)

### **Example**

**const** MongoClient = require(**'mongodb'**).MongoClient;

**const** assert = require(**'assert'**); *// Connection URL*

**const** url = **'mongodb://localhost:27017'**;*// Database Name*

**const** dbName = **'myproject'**; *// Create a new MongoClient*

**const** client = **new** MongoClient(url); *// Use connect method to connect to the Server*

client.connect(**function**(err, client) { assert.equal(**null**, err);

console.log(**"Connected correctly to server"**);

**const** db = client.db(dbName);

simplePipeline(db, **function**() { client.close(); }); });

**function** simplePipeline(db, callback)

{ **const** collection = db.collection( **'restaurants'** );

collection.aggregate( [ { **'$match'**: { **"borough"**: **"Bronx"** } }, { **'$unwind'**: **'$categories'**}, { **'$group'**: { **'\_id'**: **"$categories"**, **'Bronx restaurants'**: { **'$sum'**: 1 } } } ],

**function**(err, cursor) { assert.equal(err, **null**);

cursor.toArray(**function**(err, documents) { console.log(documents) callback(documents);

});

} );

}

|  |  |  |
| --- | --- | --- |
| **Expression** | **Description** | **Example** |
| $sum | Sums up the defined value from all documents in the collection. | db.mycol.aggregate([{$group : {\_id : "$by\_user", num\_tutorial : {$sum : "$likes"}}}]) |
| $avg | Calculates the average of all given values from all documents in the collection. | db.mycol.aggregate([{$group : {\_id : "$by\_user", num\_tutorial : {$avg : "$likes"}}}]) |
| $min | Gets the minimum of the corresponding values from all documents in the collection. | db.mycol.aggregate([{$group : {\_id : "$by\_user", num\_tutorial : {$min : "$likes"}}}]) |
| $max | Gets the maximum of the corresponding values from all documents in the collection. | db.mycol.aggregate([{$group : {\_id : "$by\_user", num\_tutorial : {$max : "$likes"}}}]) |
| $push | Inserts the value to an array in the resulting document. | db.mycol.aggregate([{$group : {\_id : "$by\_user", url : {$push: "$url"}}}]) |
| $addToSet | Inserts the value to an array in the resulting document but does not create duplicates. | db.mycol.aggregate([{$group : {\_id : "$by\_user", url : {$addToSet : "$url"}}}]) |
| $first | Gets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied “$sort”-stage. | db.mycol.aggregate([{$group : {\_id : "$by\_user", first\_url : {$first : "$url"}}}]) |
| $last | Gets the last document from the source documents according to the grouping. Typically this makes only sense together with some previously applied “$sort”-stage. | db.mycol.aggregate([{$group : {\_id : "$by\_user", last\_url : {$last : "$url"}}}]) |

## **Pipeline Concept**

In UNIX command, shell pipeline means the possibility to execute an operation on some input and use the output as the input for the next command and so on. MongoDB also supports same concept in aggregation framework. There is a set of possible stages and each of those is taken as a set of documents as an input and produces a resulting set of documents (or the final resulting JSON document at the end of the pipeline). This can then in turn be used for the next stage and so on.

Following are the possible stages in aggregation framework −

* **$project** − Used to select some specific fields from a collection.
* **$match** − This is a filtering operation and thus this can reduce the amount of documents that are given as input to the next stage.
* **$group** − This does the actual aggregation as discussed above.
* **$sort** − Sorts the documents.
* **$skip** − With this, it is possible to skip forward in the list of documents for a given amount of documents.
* **$limit** − This limits the amount of documents to look at, by the given number starting from the current positions.
* **$unwind** − This is used to unwind document that are using arrays. When using an array, the data is kind of pre-joined and this operation will be undone with this to have individual documents again. Thus with this stage we will increase the amount of documents for the next stage.

# **Replication**

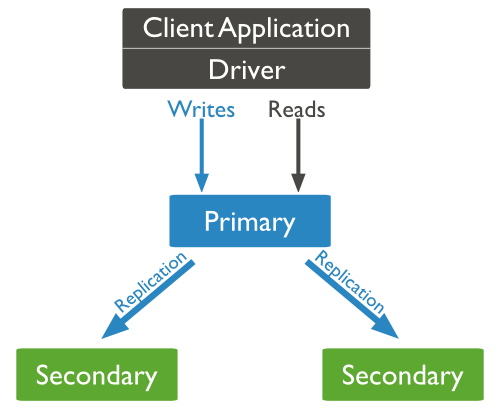
In MongoDB, Replication is the process through which we can synchronize the data of a server among multiple servers. In this way, we can provide data redundancy and increase data availability among multiple servers. The replication process always protects a MongoDB database from the loss of a server due to hardware failure or any other reason. In this way, we can provide uninterrupted availability of MongoDB data using replication servers. With the help of replication, we can ensure that the same data is always available in more than one MongoDB Server.   
So, if due to any hardware failure or any other reason, the main MongoDB server goes down, then also we can access the data from the replicated server since data has been replicated into another server at regular intervals through the replication process. Also, replication can be done for the purpose of load balancing. If we have a large number of users access the MongoDB database, then instead of connecting a single MongoDB server, we can connect users into multiple servers so that the load can be equally distributed.  
We can achieve the below advantages if we use replication in MongoDB for the production environment –

1. Using Replication, we can keep the data safe.
2. Replication process always ensure the high availability of data
3. We can take care of disaster recovery
4. No downtimes required for maintenance (like backups, index rebuilds, etc.)
5. Replica Set is always transparent to the application.

## **What is a Replica Set in MongoDB?**

In MongoDB, the replication process can be set up by creating a replica set. In MongoDB, a replica set contains multiple MongoDB servers. In this group of MongoDB servers, one server is known as a Primary Server and others are known as Secondary servers. Every secondary server always keeps copies of the primary’s data. So, if any time the primary server goes down, then the new primary server is selected from the existing secondary server and process goes on. The replication process works as below with the help of a replica set –

* Replica Set is a group of one or more standalone MongoDB Servers (normally 3 MongoDB Servers are required).
* In a Replica Set, one server is marked as Primary Server and rest are marked as a Secondary Server.
* Data writes into the Primary Server from the application first.
* Then all the data replicates to the secondary servers from the primary server.
* When the primary server is unavailable due to hardware failure or maintenance work, the election process starts to identify the new primary server and select a primary server from the secondary server lists.
* When the failed server recovered, it will again join the replica set as a secondary server.

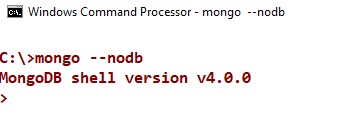


An above diagram of MongoDB replication is shown in the below image. In this image, a client application always communicates with the primary node and the primary node replicates the data to the multiple secondary nodes.

## **How to Configure Replication in MongoDB**

In this section, we will discuss how to convert a standalone MongoDB Instance into a replica set. This process is not an ideal process for the production environment. Because in production, if we need to establish a replica set then we need to provide three different MongoDB instances for the replica set. But it is a good process for gaining knowledge about the idea of replication and knowing about the configuration of replication.

**Step 1**  
Start up a mongo shell with the --nodb options from the command prompt. It will start a shell without any connection with the existing mongod instance.



**Step 2**  
Now, create a replica set with the below commands,

1. replicaSet = **new** ReplSetTest({name:'rsTest', nodes : 3})

This command instructs the shell to create a replica set with three node servers:- one primary and two secondaries.

**Step 3**  
Now run the below commands one by one to start the mongod server instances,

1. replicaSet.startSet() -- **this** command start the three mongod processes.
2. replicaSet.initiate() -- **this** command configures the replication

Now, we have three mongod processes locally on ports 20000,20001 and 20002.

**Step 4**  
Now open another command prompt and connect the mongod running on port 20000.

1. conn1 = **new** Mongo("localhost:20000")
2. connection to localhost:20000
3. rsTest: PRIMARY>

Note that, when we connect a replica set member, the prompt changes to rsTest: PRIMARY. Here PRIMARY is the state of the member and rsTest is the identifier of the replica set.

Now, if we want to check that the mongod instance is actually is a primary node or not, then we need to run the below command to check the status of the replica set –

1. primaryDB.isMaster()

# **Sharding**

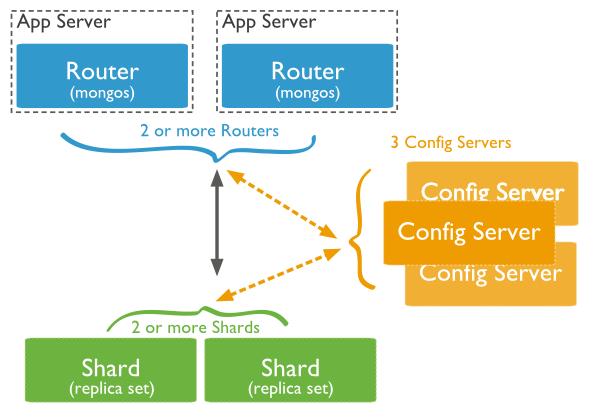
Sharding is the process of storing data records across multiple machines and it is MongoDB's approach to meeting the demands of data growth. As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput. Sharding solves the problem with horizontal scaling. With sharding, you add more machines to support data growth and the demands of read and write operations.

## Why Sharding?

* In replication, all writes go to master node
* Latency sensitive queries still go to master
* Single replica set has limitation of 12 nodes
* Memory can't be large enough when active dataset is big
* Local disk is not big enough
* Vertical scaling is too expensive

## **Sharding in MongoDB**

The following diagram shows the Sharding in MongoDB using sharded cluster.



In the following diagram, there are three main components −

* **Shards** − Shards are used to store data. They provide high availability and data consistency. In production environment, each shard is a separate replica set.
* **Config Servers** − Config servers store the cluster's metadata. This data contains a mapping of the cluster's data set to the shards. The query router uses this metadata to target operations to specific shards. In production environment, sharded clusters have exactly 3 config servers.
* **Query Routers** − Query routers are basically mongo instances, interface with client applications and direct operations to the appropriate shard. The query router processes and targets the operations to shards and then returns results to the clients. A sharded cluster can contain more than one query router to divide the client request load. A client sends requests to one query router. Generally, a sharded cluster have many query routers.

## **Step by Step Sharding Cluster Example**

**Step 1)** Create a separate database for the config server.

mkdir /data/configdb

**Step 2)** Start the mongodb instance in configuration mode. Suppose if we have a server named Server D which would be our configuration server, we would need to run the below command to configure the server as a configuration server.

mongod –configdb ServerD: 27019

**Step 3)** Start the mongos instance by specifying the configuration server

mongos –configdb ServerD: 27019

**Step 4)** From the mongo shell connect to the mongo's instance

mongo –host ServerD –port 27017

**Step 5)** If you have Server A and Server B which needs to be added to the cluster, issue the below commands

sh.addShard("ServerA:27017")sh.addShard("ServerB:27017")

**Step 6)** Enable sharding for the database. So if we need to shard the Employeedb database, issue the below command

sh.enableSharding(Employeedb)

**Step 7)** Enable sharding for the collection. So if we need to shard the Employee collection, issue the below command

Sh.shardCollection("db.Employee" , { "Employeeid" : 1 , "EmployeeName" : 1})