WEEK 9

Introduction NoSQL – 1

A database Management System provides the mechanism to store and retrieve the data. There are different kinds of database Management Systems:

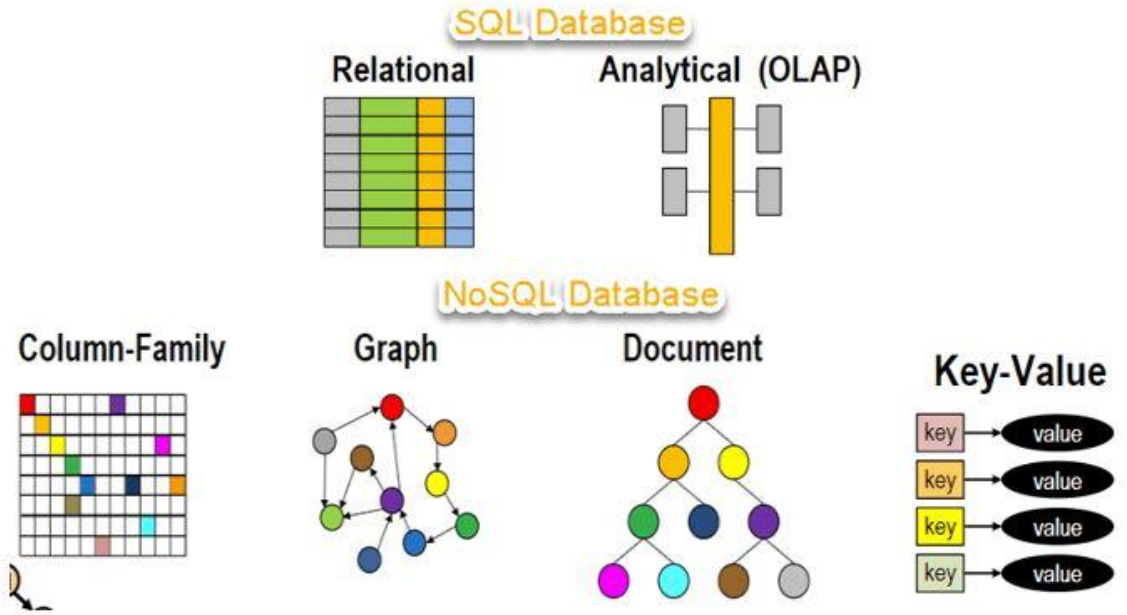
1. RDBMS (Relational Database Management Systems)
2. OLAP (Online Analytical Processing)
3. NoSQL (Not only SQL)

What is NoSQL?

NoSQL Database is a non-relational Data Management System, that does not require a fixed schema. It avoids joins, and is easy to scale. The major purpose of using a NoSQL database is for distributed data stores with humongous data storage needs. NoSQL is used for Big data and real-time web apps. For example, companies like Twitter, Facebook and Google collect terabytes of user data every single day.

NoSQL database stands for "Not Only SQL" or "Not SQL." Though a better term would be "NoREL", NoSQL caught on. Carl Strozz introduced the NoSQL concept in 1998.

Traditional RDBMS uses SQL syntax to store and retrieve data for further insights. Instead, a NoSQL database system encompasses a wide range of database technologies that can store structured, semi-structured, unstructured and polymorphic data. Let's understand about NoSQL with a diagram in this NoSQL database tutorial:



Brief History of NoSQL Databases

- 1998- Carlo Strozzi use the term NoSQL for his lightweight, open-source relational database
- 2000- Graph database Neo4j is launched
- 2004- Google BigTable is launched
- 2005- CouchDB is launched
- 2007- The research paper on Amazon Dynamo is released
- 2008- Facebooks open sources the Cassandra project
- 2009- The term NoSQL was reintroduced

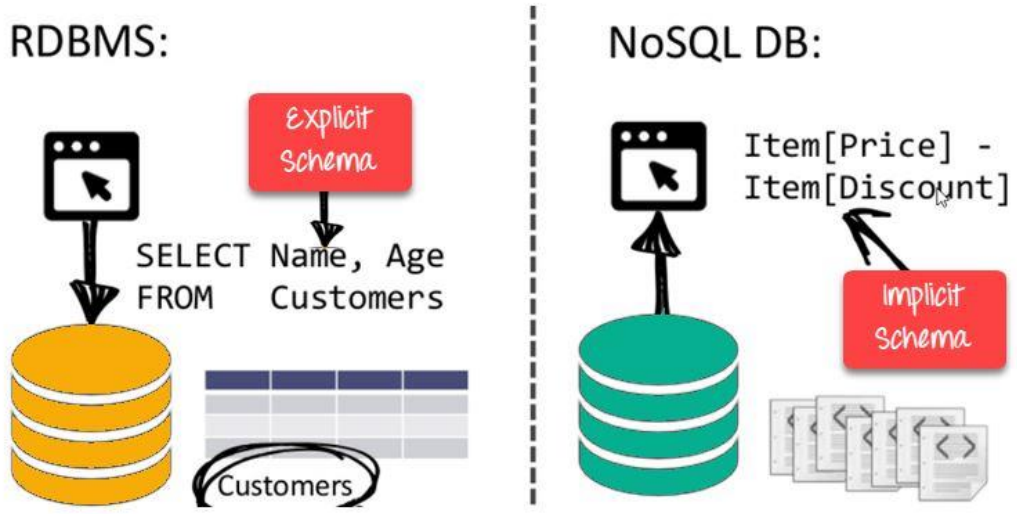
Features of NoSQL

Non-relational

- NoSQL databases never follow the relational model
- Never provide tables with flat fixed-column records
- Work with self-contained aggregates or BLOBs
- Doesn't require object-relational mapping and data normalization
- No complex features like query languages, query planners, referential integrity joins, ACID

Schema-free

- NoSQL databases are either schema-free or have relaxed schemas
- Do not require any sort of definition of the schema of the data
- Offers heterogeneous structures of data in the same domain



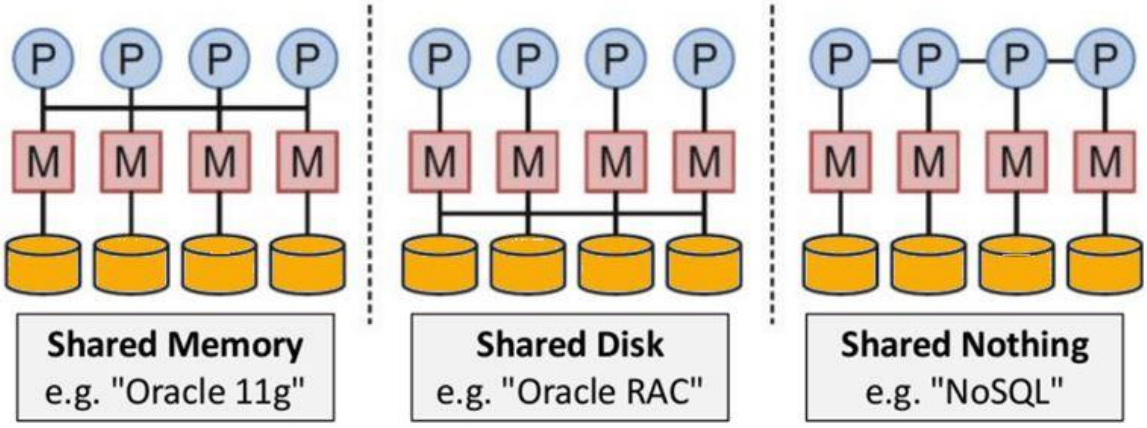
NoSQL is Schema-Free

Simple API

- Offers easy to use interfaces for storage and querying data provided
- APIs allow low-level data manipulation & selection methods
- Text-based protocols mostly used with HTTP REST with JSON
- Mostly used no standard based NoSQL query language
- Web-enabled databases running as internet-facing services

Distributed

- Multiple NoSQL databases can be executed in a distributed fashion
- Offers auto-scaling and fail-over capabilities
- Often ACID concept can be sacrificed for scalability and throughput
- Mostly no synchronous replication between distributed nodes Asynchronous Multi-Master Replication, peer-to-peer, HDFS Replication
- Only providing eventual consistency
- Shared Nothing Architecture. This enables less coordination and higher distribution.



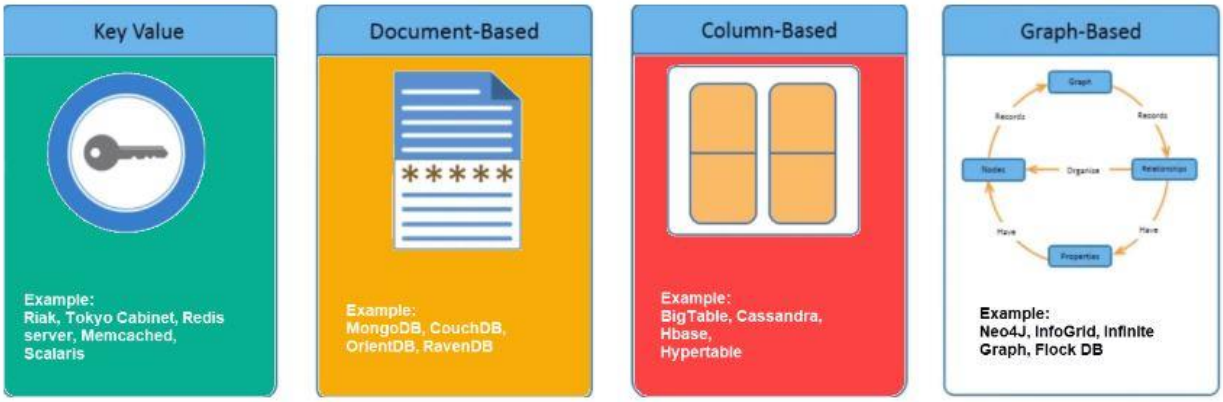
NoSQL is Shared Nothing.

Types of NoSQL Databases

NoSQL Databases are mainly categorized into four types: Key-value pair, Column-oriented, Graph-based and Document-oriented. Every category has its unique attributes and limitations. None of the above-specified database is better to solve all the problems. Users should select the database based on their product needs.

Types of NoSQL Databases:

- Key-value Pair Based
- Column-oriented Graph
- Graphs based
- Document-oriented



Key Value Pair Based

Data is stored in key/value pairs. It is designed in such a way to handle lots of data and heavy load.

Key-value pair storage databases store data as a hash table where each key is unique, and the value can be a JSON, BLOB(Binary Large Objects), string, etc.

For example, a key-value pair may contain a key like “Website” associated with a value like “Sanpoly”.

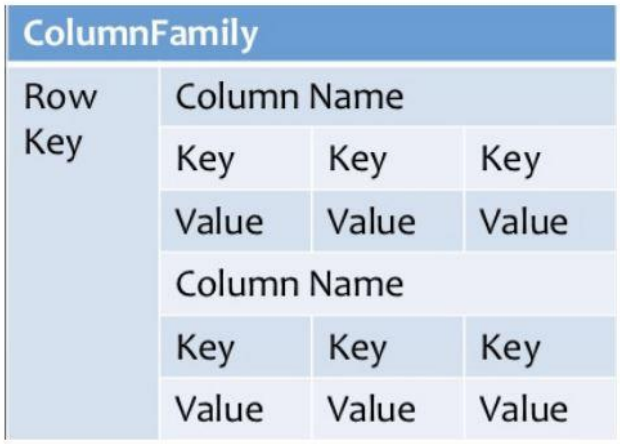
Key	Value
Name	Joe Bloggs
Age	42
Occupation	Stunt Double
Height	175cm
Weight	77kg

It is one of the most basic NoSQL database example. This kind of NoSQL database is used as a collection, dictionaries, associative arrays, etc. Key value stores help the developer to store schema-less data. They work best for shopping cart contents.

Redis, Dynamo, Riak are some NoSQL examples of key-value store DataBases. They are all based on Amazon’s Dynamo paper.

Column-based

Column-oriented databases work on columns and are based on BigTable paper by Google. Every column is treated separately. Values of single column databases are stored contiguously.



Column based NoSQL database

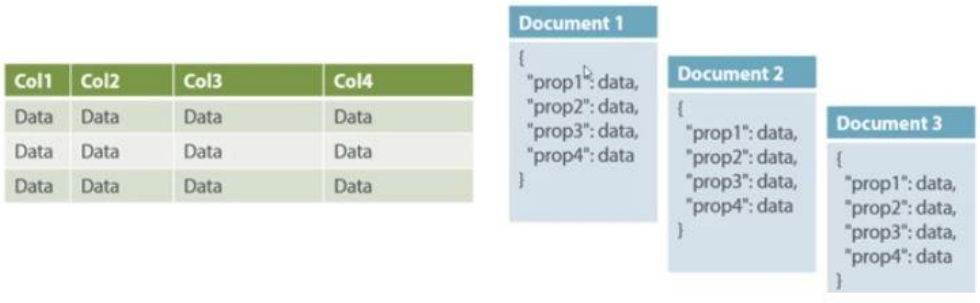
They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column.

Column-based NoSQL databases are widely used to manage data warehouses, business intelligence, CRM, Library card catalogs,

HBase, Cassandra, HBase, Hypertable are NoSQL query examples of column based database.

Document-Oriented:

Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document. The document is stored in JSON or XML formats. The value is understood by the DB and can be queried.



Relational Vs. Document

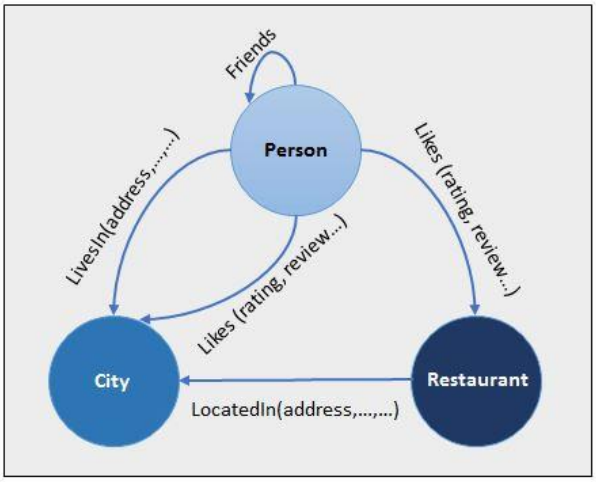
In this diagram on your left you can see we have rows and columns, and in the right, we have a document database which has a similar structure to JSON. Now for the relational database, you have to know what columns you have and so on. However, for a document database, you have data store like JSON object. You do not require to define which make it flexible.

The document type is mostly used for CMS systems, blogging platforms, real-time analytics & e-commerce applications. It should not use for complex transactions which require multiple operations or queries against varying aggregate structures.

Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes, MongoDB, are popular Document originated DBMS systems.

Graph-Based

A graph type database stores entities as well the relations amongst those entities. The entity is stored as a node with the relationship as edges. An edge gives a relationship between nodes. Every node and edge has a unique identifier.



Compared to a relational database where tables are loosely connected, a Graph database is a multi-relational in nature. Traversing relationship is fast as they are already captured into the DB, and there is no need to calculate them.

Graph base database mostly used for social networks, logistics, spatial data.

Neo4J, Infinite Graph, OrientDB, FlockDB are some popular graph-based databases.

What is the CAP Theorem?

CAP theorem is also called brewer's theorem. It states that is impossible for a distributed data store to offer more than two out of three guarantees

1. Consistency
2. Availability
3. Partition Tolerance

Consistency:

The data should remain consistent even after the execution of an operation. This means once data is written, any future read request should contain that data. For example, after updating the order status, all the clients should be able to see the same data.

Availability:

The database should always be available and responsive. It should not have any downtime.

Partition Tolerance:

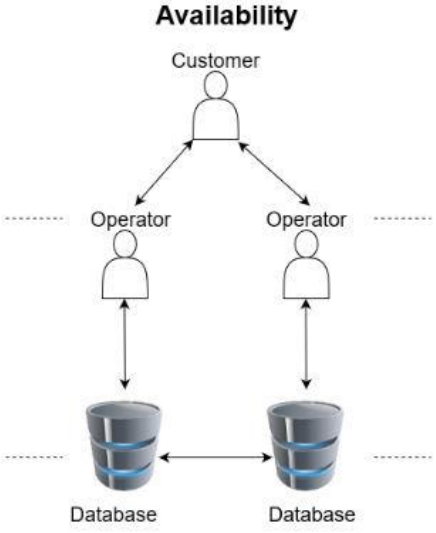
Partition Tolerance means that the system should continue to function even if the communication among the servers is not stable. For example, the servers can be partitioned into multiple groups which may not communicate with each other. Here, if part of the database is unavailable, other parts are always unaffected.

Understanding CAP theorem with an Example

Availability

Imagine there is a very popular mobile operator in your city and you are its customer because of the amazing plans it offers. Besides that, they also provide an amazing customer care service where you can call anytime and get your queries and concerns answered quickly and efficiently. Whenever a customer calls them, the mobile operator is able to connect them to one of their customer care operators.

The customer is able to elicit any information required by her/him about his accounts like balance, usage, or other information. We call this **Availability** because every customer is able to connect to the operator and get the information about the user/customer.

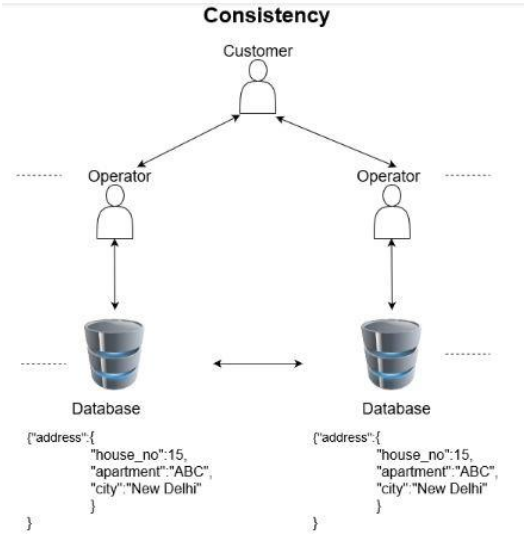


Consistency

Now, you have recently shifted to a new house in the city and you want to update your address registered with the mobile operator. You decide to call the customer care operator and update it with them. When you call, you connect with an operator. This operator makes the relevant changes in the system. But once you have put down the phone, you realize you told them the correct street name but the old house number (old habits die hard!).

So, you frantically call the customer care again. This time when you call, you connect with a different customer care operator but they are able to access your records as well and know that you have recently updated your address. They make the relevant changes in the house number and the rest of the address is the same as the one you told the last operator.

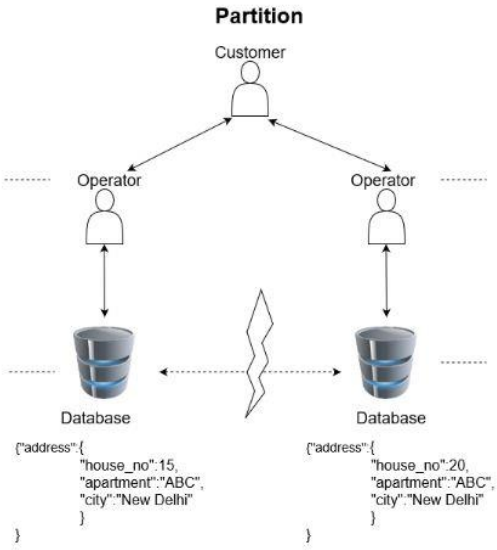
We call this as **Consistency** because even though you connect to a different customer care operator, they were able to retrieve the same information.



Partition tolerance

Recently you have noticed that your current mobile plan does not suit you. You do not access that much mobile data any longer because you have good wi-fi facilities at home and at the office, and you hardly step outside anywhere. Therefore, you want to update your mobile plan. So you decide to call the customer care once again.

On connecting with the operator this time, they tell you that they have not been able to update their records due to some issues. So the information lying with the operator might not be up to date, therefore they cannot update the information. We can say here that the service is broken or there is no **Partition tolerance.**



The CAP theorem states that a distributed database system has to make a tradeoff between Consistency and Availability when a Partition occurs.

A distributed database system is bound to have partitions in a real-world system due to network failure or some other reason. Therefore, partition tolerance is a property we cannot avoid while building our system. So, a distributed system will either choose to give up on Consistency or Availability but not on Partition tolerance.

For example, in a distributed system, if a partition occurs between two nodes, it is impossible to provide consistent data on both the nodes and availability of complete data. Therefore, in such a scenario we either choose to compromise on Consistency or on Availability. Hence, a NoSQL distributed database is either characterized as CP or AP. CA type databases are generally the monolithic databases that work on a single node and provide no distribution. Hence, they require no partition tolerance.

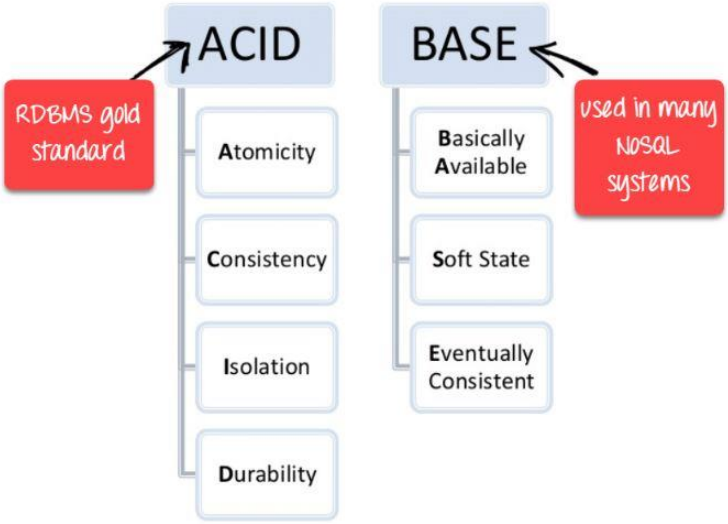
Eventual Consistency

The term “eventual consistency” means to have copies of data on multiple machines to get high availability and scalability. Thus, changes made to any data item on one machine has to be propagated to other replicas.

Data replication may not be instantaneous as some copies will be updated immediately while others in due course of time. These copies may be mutually, but in due course of time, they become consistent. Hence, the name eventual consistency.

BASE: **B**asically **A**vailable, **S**oft state, **E**ventual consistency

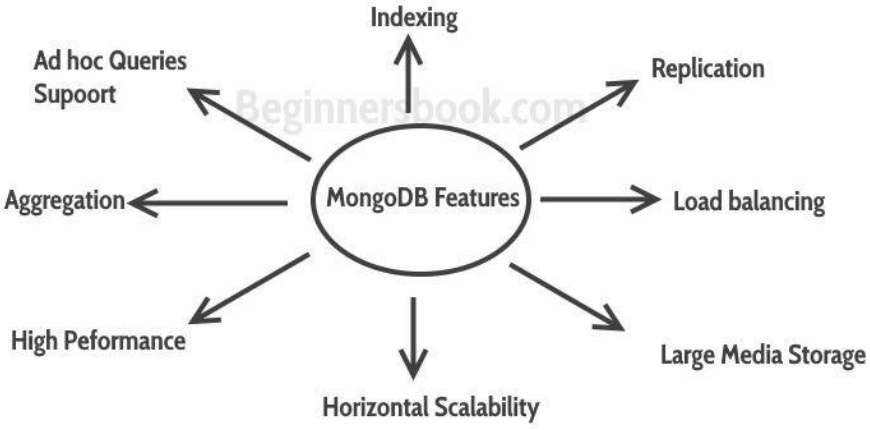
- Basically, available means DB is available all the time as per CAP theorem
- Soft state means even without an input; the system state may change
- Eventual consistency means that the system will become consistent over time.



What is MongoDB?

- **MongoDB** is a document-oriented NoSQL database used for high volume data storage. Instead of using tables and rows as in the traditional relational databases, MongoDB makes use of collections and documents.
- Documents consist of key-value pairs which are the basic unit of data in MongoDB. Collections contain sets of documents and function which is the equivalent of relational database tables. MongoDB is a database which came into light around the mid-2000s.

MongoDB Features



1. MongoDB provides **high performance**. Most of the operations in the MongoDB are faster compared to relational databases.
2. MongoDB provides **auto replication** feature that allows you to quickly recover data in case of a failure.
3. Horizontal scaling is possible in MongoDB because of sharing. Sharding is partitioning of data and placing it on multiple machines in such a way that the order of the data is preserved.

Horizontal scaling vs vertical scaling:

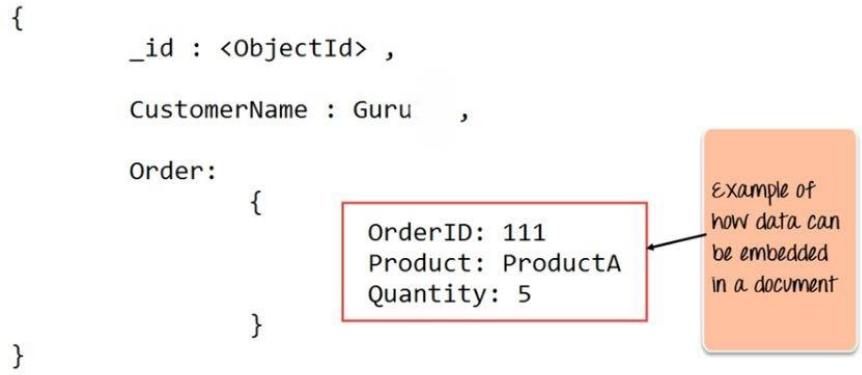
Vertical scaling means adding more resources to the existing machine while horizontal scaling means adding more machines to handle the data. Vertical scaling is not that easy to implement, on the other hand horizontal scaling is easy to implement. Horizontal scaling database examples: MongoDB, Cassandra etc.

Load balancing: Horizontal scaling allows MongoDB to balance the load. **High Availability**: Auto Replication improves the availability of MongoDB database. **Indexing:** Index is a single field within the document. Indexes are used to quickly locate data without having to search every document in a MongoDB database. This improves the performance of operations performed on the MongoDB database.

MongoDB Example

The below example shows how a document can be modeled in MongoDB.

1. The _id field is added by MongoDB to uniquely identify the document in the collection.
2. What you can note is that the Order Data (OrderID, Product, and Quantity) which in RDBMS will normally be stored in a separate table, while in MongoDB it is actually stored as an embedded document in the collection itself. This is one of the key differences in how data is modeled in MongoDB.



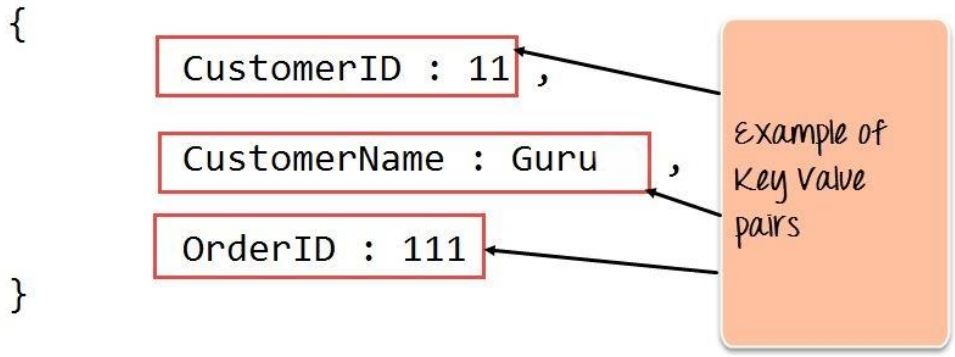
Key Components of MongoDB Architecture

Below are a few of the common terms used in MongoDB

1. **_id** – This is a field required in every MongoDB document. The _id field represents a unique value in the MongoDB document. The _id field is like the document's primary key. If you create a new document without an _id field, MongoDB will automatically create the field. So, for example, if we see the example of the above customer table, Mongo DB will add a 24-digit unique identifier to each document in the collection.

_Id	CustomerID	CustomerName	OrderID
563479cc8a8a4246bd27d784	11	Guru	111
563479cc7a8a4246bd47d784	22	Trevor Smith	222
563479cc9a8a4246bd57d784	33	Nicole	333

2. **Collection** – This is a grouping of MongoDB documents. A collection is the equivalent of a table which is created in any other RDMS such as Oracle or MS SQL. A collection exists within a single database. As seen from the introduction collections don't enforce any sort of structure.
3. **Cursor** – This is a pointer to the result set of a query. Clients can iterate through a cursor to retrieve results.
4. **Database** – This is a container for collections like in RDMS wherein it is a container for tables. Each database gets its own set of files on the file system. A MongoDB server can store multiple databases.
5. **Document** – A record in a MongoDB collection is basically called a document. The document, in turn, will consist of field name and values.
6. **Field** – A name-value pair in a document. A document has zero or more fields. Fields are analogous to columns in relational databases. The following diagram shows an example of Fields with Key value pairs. So, in the example below CustomerID and 11 is one of the key value pair's defined in the document.



7. **JSON** – This is known as JavaScript Object Notation. This is a human-readable, plain text format for expressing structured data. JSON is currently supported in many programming languages.

Data Modelling in MongoDB

As we know that the data in MongoDB has a flexible schema. Unlike in SQL databases, where you must have a table's schema declared before inserting data, MongoDB's collections do not enforce document structure. This sort of flexibility is what makes MongoDB so powerful.

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 –

```
{
        _id: ,
        Emp_ID: "10025AE336"
        Personal_details:{
                First_Name: "Radhika",
                Last_Name: "Sharma",
                Date_Of_Birth: "1995-09-26"
        },
        Contact: {
                e-mail: "radhika_sharma.123@gmail.com",
                phone: "9848022338"
        },
        Address: {
                city: "Hyderabad",
                Area: "Madapur",
                State: "Telangana"
        }
}
```

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:

Employee:

```
{
        _id: <ObjectId101>,
        Emp_ID: "10025AE336"
}
```

Personal_details:

```
{
        _id: <ObjectId102>,
        empDocID: " ObjectId101",
        First_Name: "Radhika",
        Last_Name: "Sharma",
        Date_Of_Birth: "1995-09-26"
}
```

Contact:

```
{
        _id: <ObjectId103>,
        empDocID: " ObjectId101",
        e-mail: "radhika_sharma.123@gmail.com",
        phone: "9848022338"
}
```

Address:

```
{
        _id: <ObjectId104>,
        empDocID: " ObjectId101",
        city: "Hyderabad",
        Area: "Madapur",
        State: "Telangana"
}
```

Considerations while designing Schema in MongoDB

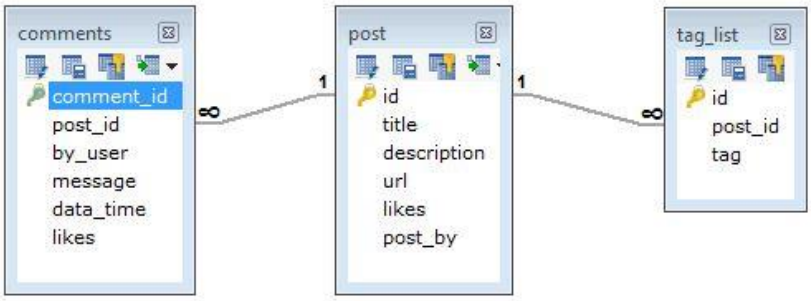
- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while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

Example

Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.

- Every post has the 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.
- Every post has comments given by users along with their name, message, data-time and likes.
- On each post, there can be zero or more comments.

In RDBMS schema, design for above requirements will have minimum three tables.



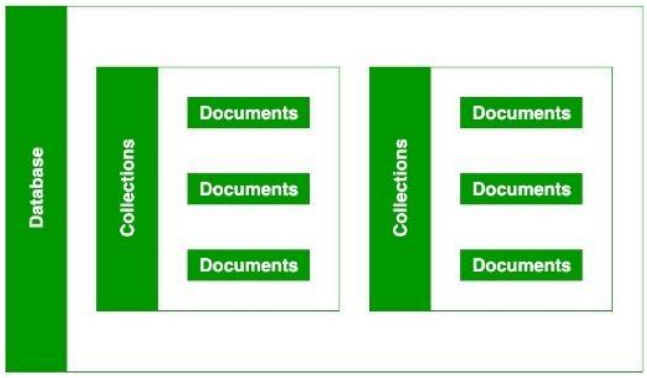
While in MongoDB schema, design will have one collection post and the following structure –

```
{
  _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: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    }
  ]
}
```

So while showing the data, in RDBMS you need to join three tables and in MongoDB, data will be shown from one collection only.

Working with MongoDB

As we know that MongoDB is a database server and the data is stored in these databases. Or in other words, MongoDB environment gives you a server that you can start and then create multiple databases on it using MongoDB.
Because of its NoSQL database, the data is stored in the collections and documents. Hence the database, collection, and documents are related to each other as shown below:



- The MongoDB database contains collections just like the MYSQL database contains tables. You are allowed to create multiple databases and multiple collections.
- Now inside of the collection we have documents. These documents contain the data we want to store in the MongoDB database and a single collection can contain multiple documents and you are schema-less means it is not necessary that one document is similar to another.
- The documents are created using the fields. Fields are key-value pairs in the documents, it is just like columns in the relation database. The value of the fields can be of any BSON data types like double, string, boolean, etc.
- The data stored in the MongoDB is in the format of BSON documents. Here, BSON stands for Binary representation of JSON documents. Or in other words, in the backend, the MongoDB server converts the JSON data into a binary form that is known as BSON and this BSON is stored and queried more efficiently.
- In MongoDB documents, you are allowed to store nested data. This nesting of data allows you to create complex relations between data and store them in the same document which makes the working and fetching of data extremely efficient as compared to SQL. In SQL, you need to write complex joins to get the data from table 1 and table 2. The maximum size of the BSON document is 16MB.

MongoDB Shell – mongosh

MongoDB Shell is the quickest way to connect, configure, query, and work with your MongoDB database. It acts as a command-line client of the MongoDB server.

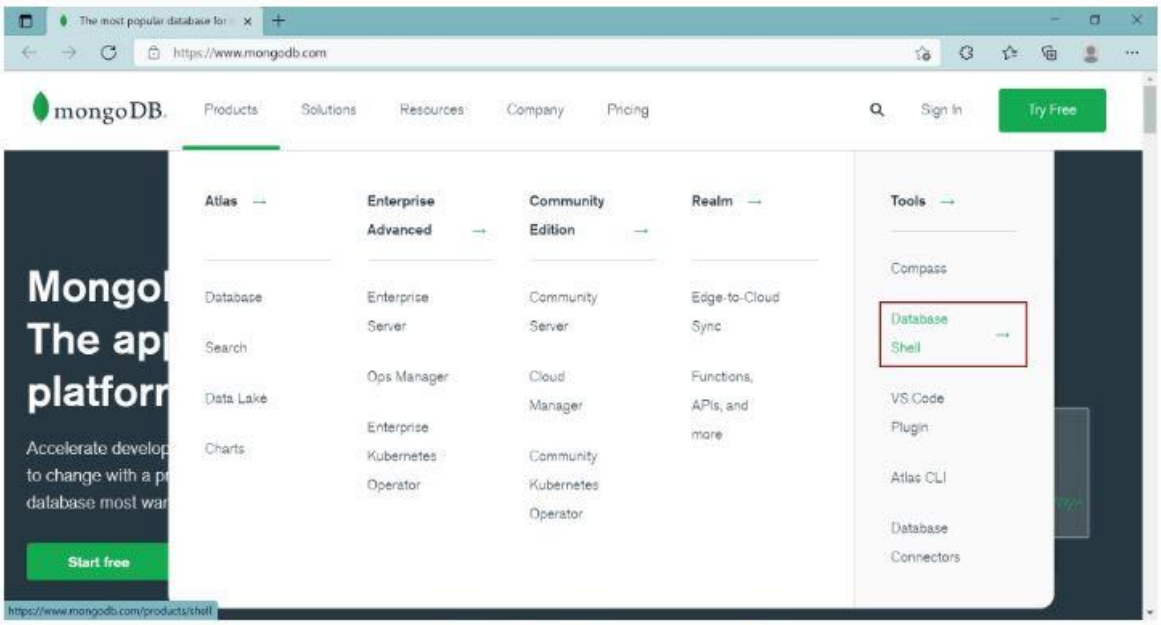
The MongoDB Shell is a standalone, open-source product and developed separately from the MongoDB Server under the Apache 2 license. It is a fully functional JavaScript and Node.js 14.x REPL for interacting with MongoDB servers.

In the installation folder, if you find mongosh.exe instead of mongo.exe then you already have a new MongoDB shell. If you don't find it, then you need to install it separately.

The new MongoDB Shell mongosh has some more features than old shell mongo such as intelligent autocomplete and syntax highlighting, easy to understand error messages, formatting feature to present the output in a readable format, etc. However, all the commands will be executed in mongosh as well as mongo shell.

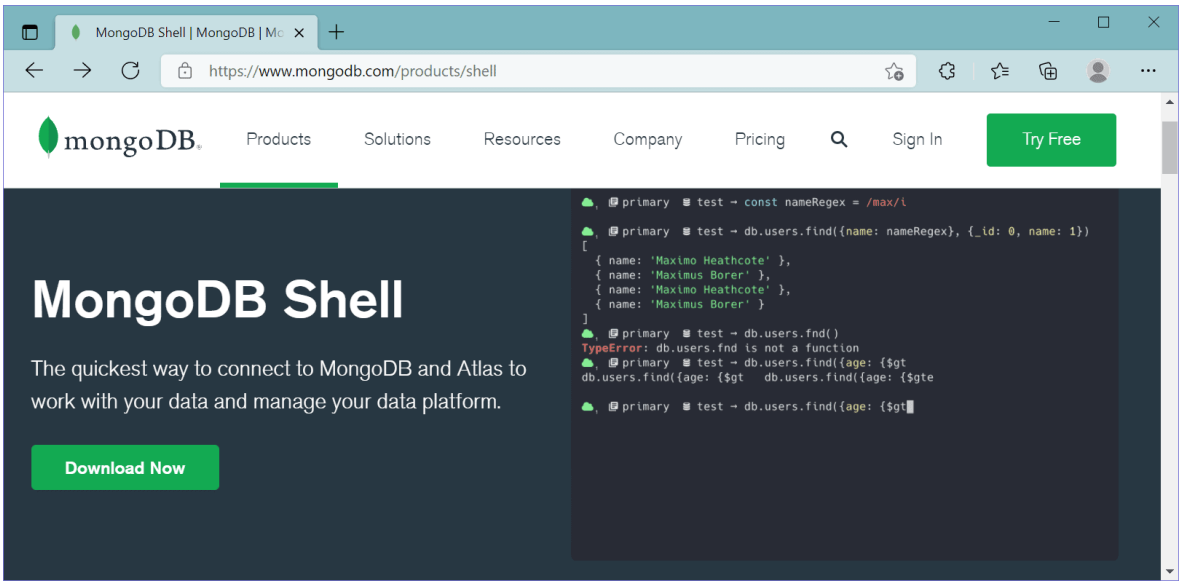
Install mongosh

To install the new MongoDB shell (mongosh), visit www.mongodb.com and click on Product menu -> Tools -> Database Shell, as shown below.



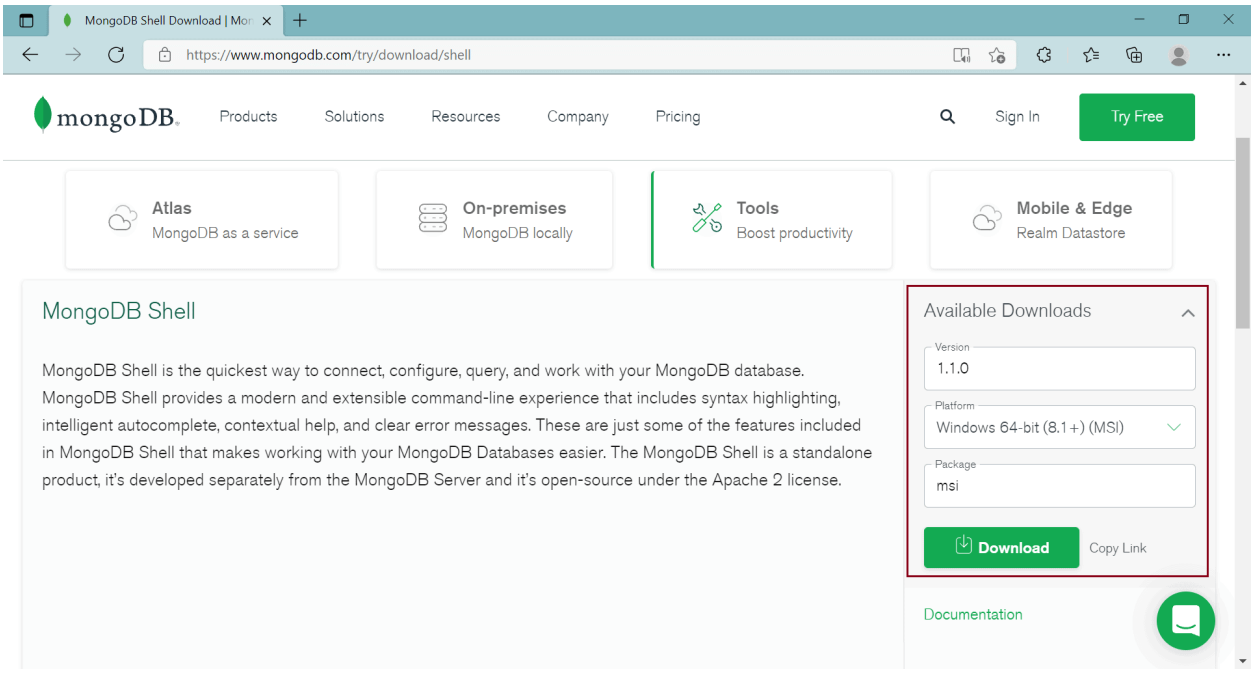
Download New MongoDB Shell

On the MongoDB Shell page, click on the Download button to download the shell.

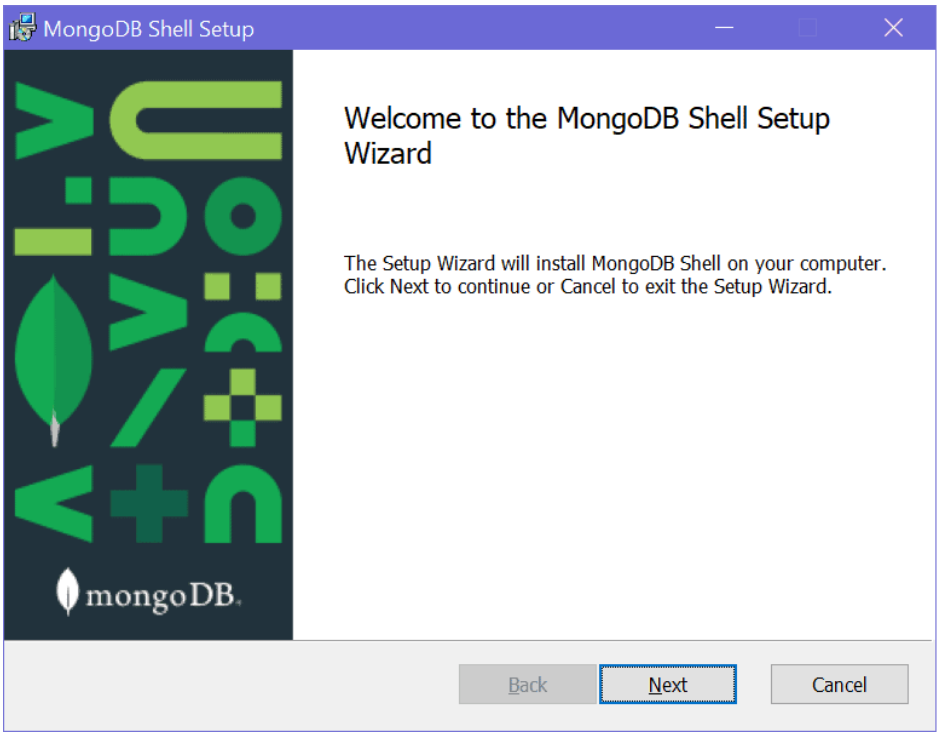


Download New MongoDB Shell

This will take you to a page where you can select a version, platform, and package to download, as shown below.

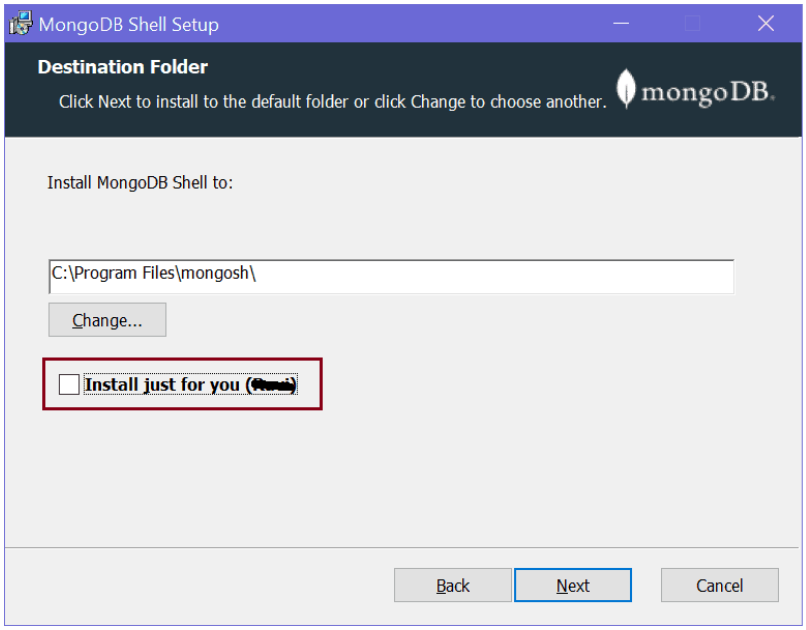


Click on the Download button to download the installer file.
Now, click on the downloaded installer file to start the installation wizard, as shown below.



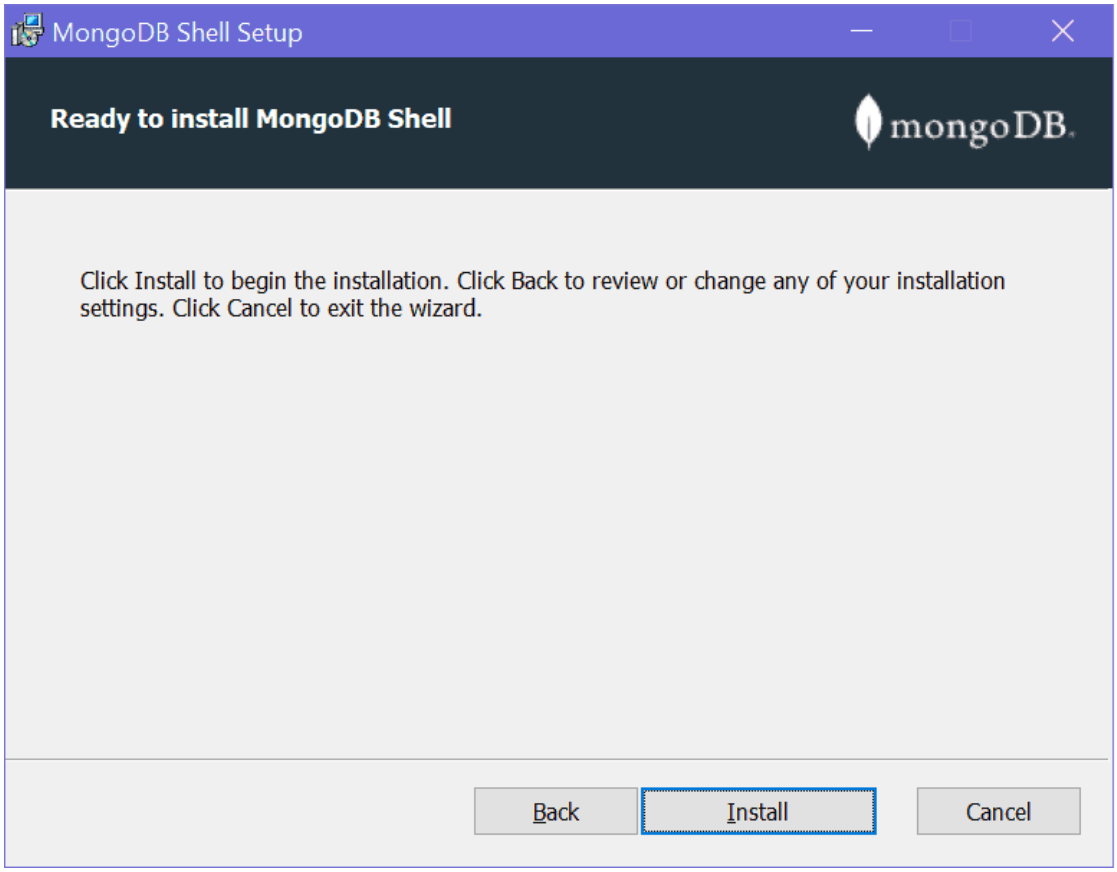
Install New MongoDB Shell - mongosh

Click Next to go to next step shown below.



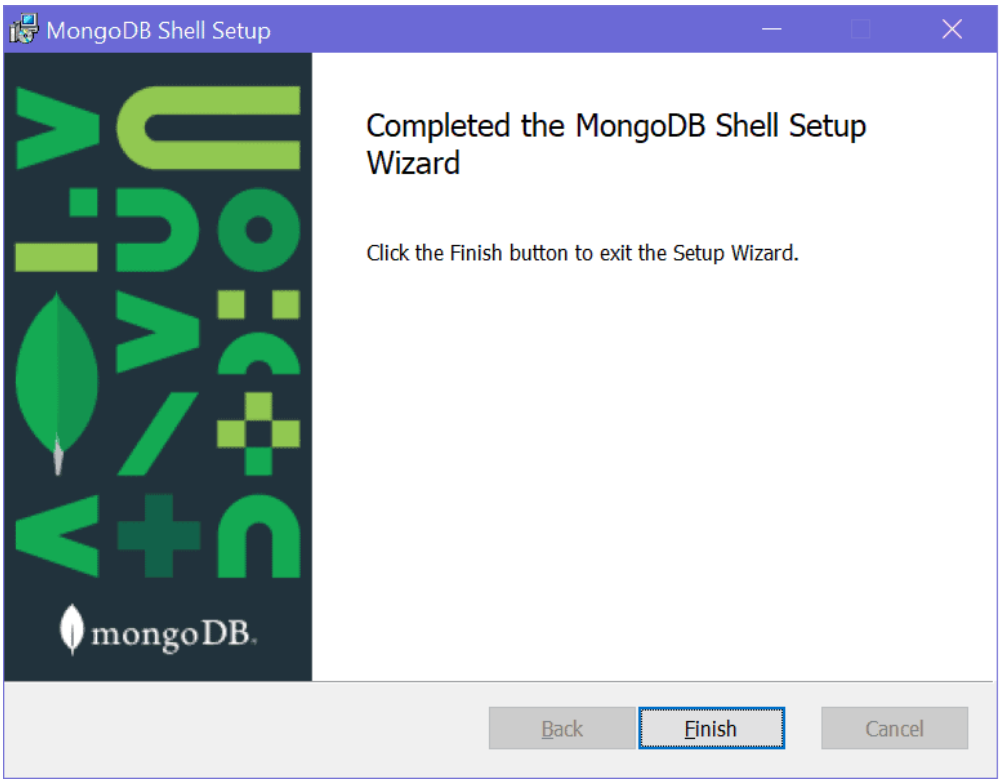
Install New MongoDB Shell - mongosh

Here, uncheck the checkbox if you want to install shell for all users on your local machine and click Next.



Install New MongoDB Shell - mongosh

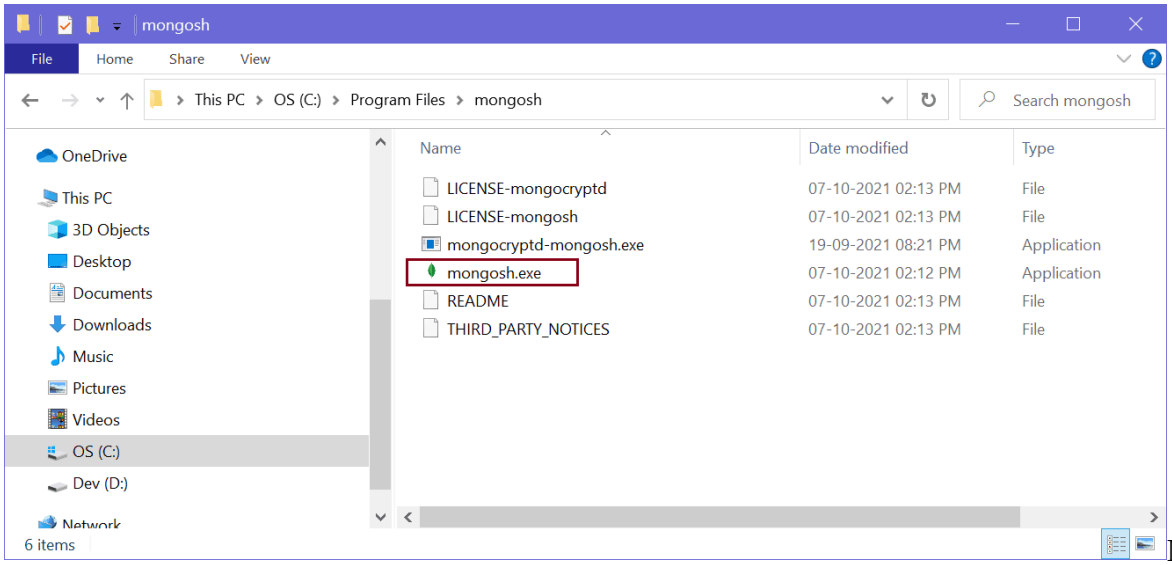
Click on the Install button to start the installation. It should quickly install it.



Install New MongoDB Shell - mongosh

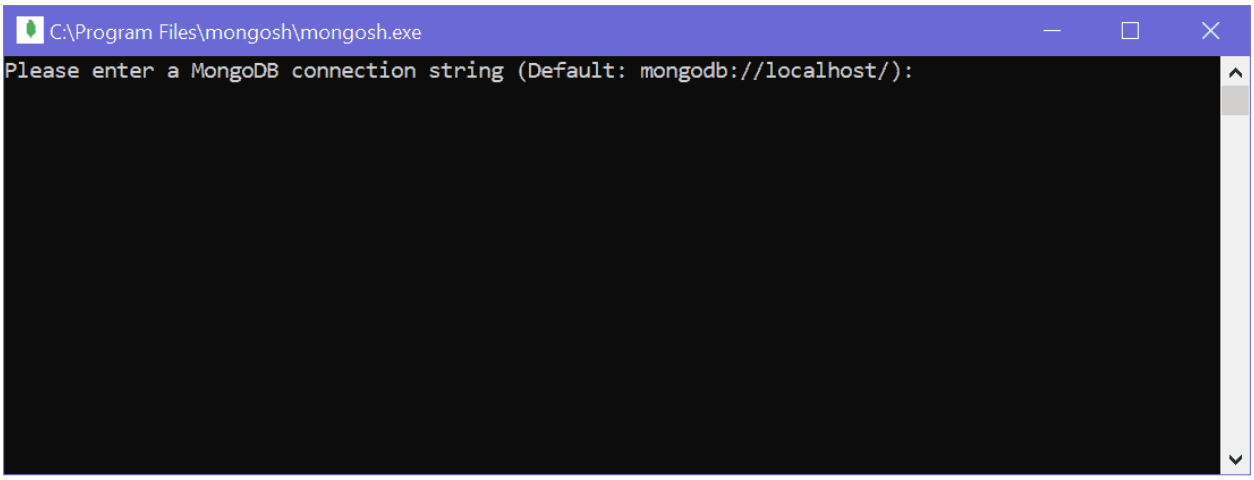
Once installation completes, click the Finish button to close the wizard.

This should have installed mongosh in "C:\Program Files\mongosh" folder on Windows, as shown below.



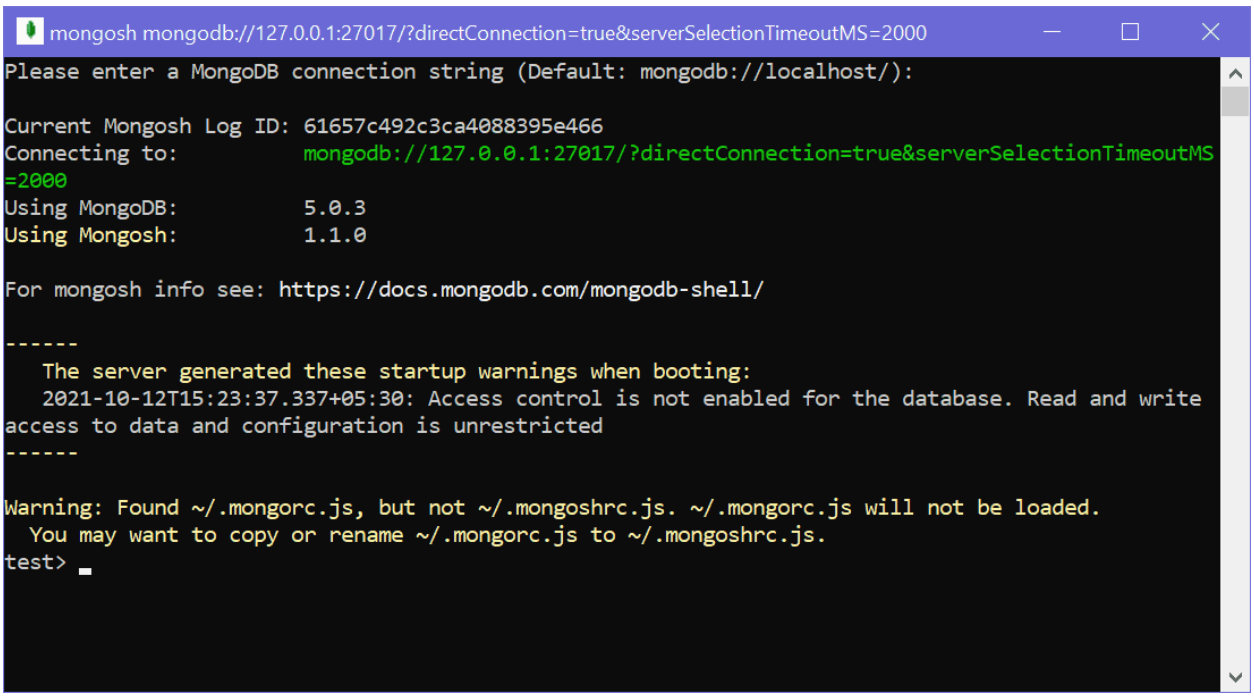
New MongoDB Shell - mongosh

Click on the mongosh.exe to open a new MongoDB shell, as shown below.



New MongoDB Shell - mongosh

Press Enter to start the shell, as shown below.



New MongoDB Shell - mongosh

Alternatively, open a new command prompt on Windows and write mongosh and press Enter. It will open the same MongoDB shell.

Execute MongoDB Commands

You can execute MongoB commands for CRUD operations on MongoDB shell (mongo or mongosh). For example, execute the "shows dbs" command to see all the databases on the connected MongoDB server.

```
> show dbs
admin 41 kB
config 111 kB
local 41 kB
```

Use the "db" command to check the current database.

```
> db
test
```

Run the .editor command to execute multi-line commands. Press Ctrl + d to run a command or Ctrl + c to cancel.

MongoDB shell is JavaScript and Node.js REPL, so you can also execute limited JavaScript code.

```
> "Hello".length
5
```

Press Ctrl + c twice to exit from the MongoDB shell.

MongoDB Compass

MongoDB Compass is a GUI for MongoDB. It is also known as MongoDB GUI. MongoDB allows users to analyze the content of their stored data without any prior knowledge of MongoDB query syntax. When we explore exploring our data in the visual environment, we can use Compass GUI to optimize performance, manage indexes, and implement document-validation.

All the versions of MongoDB

Compass are opensource (i.e., we can freely deploy and view the repositories of all MongoDB GUI versions). The source repositories of MongoDB compass can be found on the following link of GitHub https://github.com/mongodb-js/compass/

Available Compass Edition

MongoDB GUI is available in the following four editions:

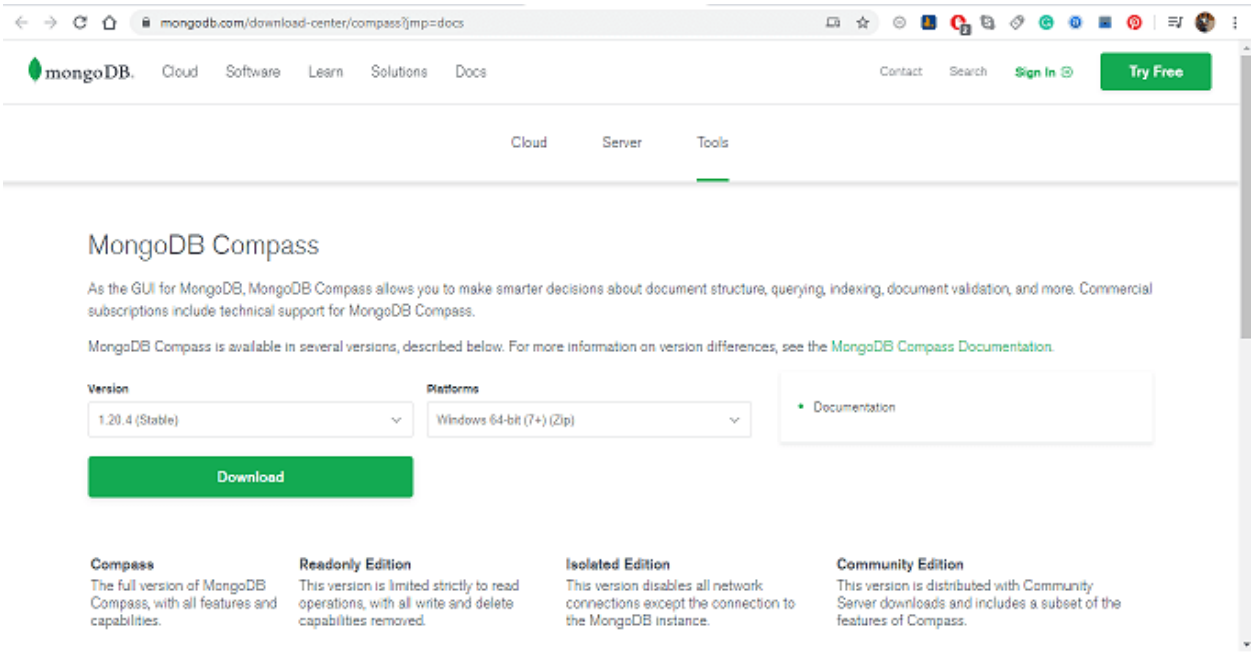
- **Compass Community:** This edition is designed for developing with MongoDB and includes a subset of the features of Compass.
- **Compass:** It is released as the full version of MongoDB Compass. It includes all the features and capabilities that MongoDB provides.
- **Compass Randomly:** It is limited to read operation only with all update and delete capabilities removed.
- **Compass Isolated:** The Isolated edition of MongoDB compass doesn't start any network requests except to the MongoDB server to which MongoDB GUI connects. It is designed to use in highly secure environments.

How to Download and Install MongoDB Compass

Step 1: To download MongoDB Compass, you can use your preferred web browser, and Open the https://www.mongodb.com/download-center/compass?jmp=docs page.

Step 2: You need to select the installer and version you prefer. GUI installer are available as a .exe or .msi package or a .zip archive.

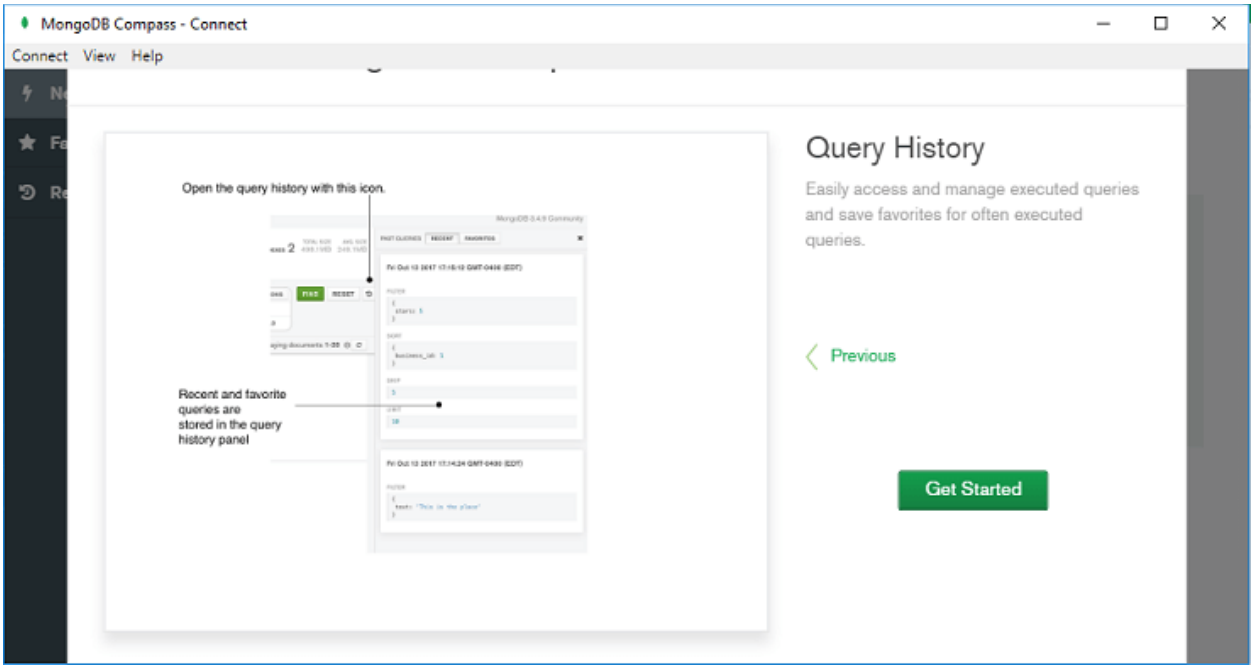
Step 3: Finally, Click on the download button.



Step 4: Click on the installer file after the download is complete.

Step 5: Follow the pop-ups to install MongoDB Compass GUI.

Step 6: Once it is installed, it launches and ask you to configure privacy settings and specify update preference.

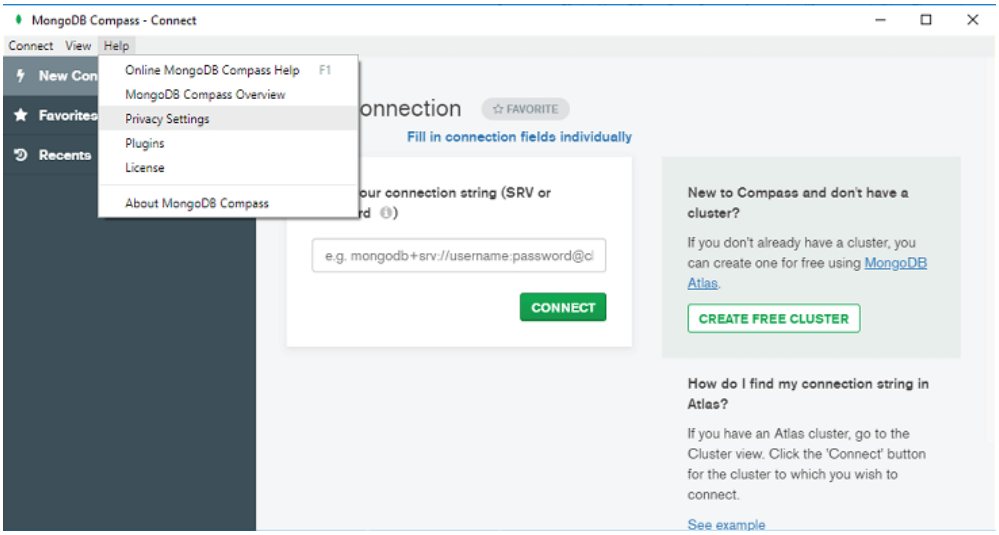


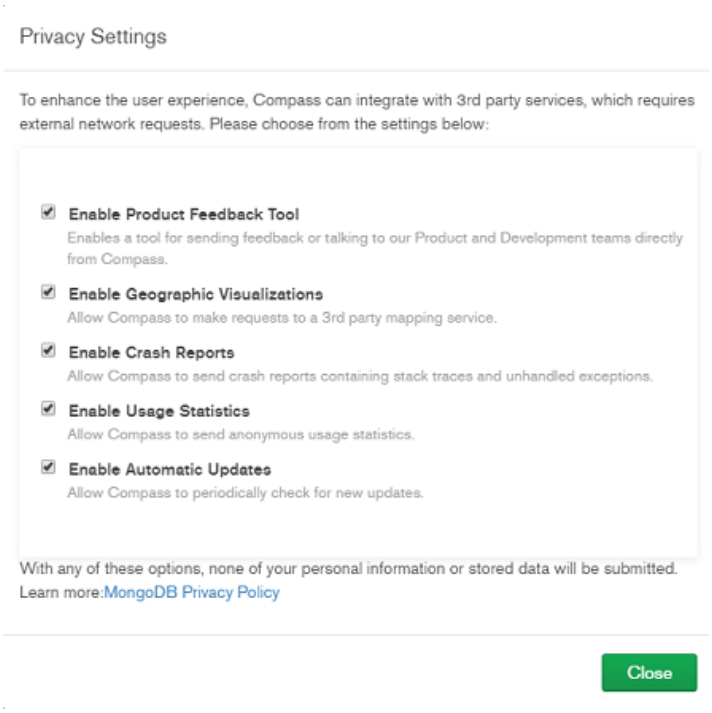
Update MongoDB Compass

There are two ways we can use the updated version of MongoDB Compass

1. We can download and install the latest released version of MongoDB GUI from the official website of MongoDB at any time. We need to check the S/W and H/W requirements for our OS and required version of MongoDB compass to ensure Compass GUI is compatible with our system.

2. We can update Compass by enabling the automatic updates from the Help -> Privacy Settings as shown below.





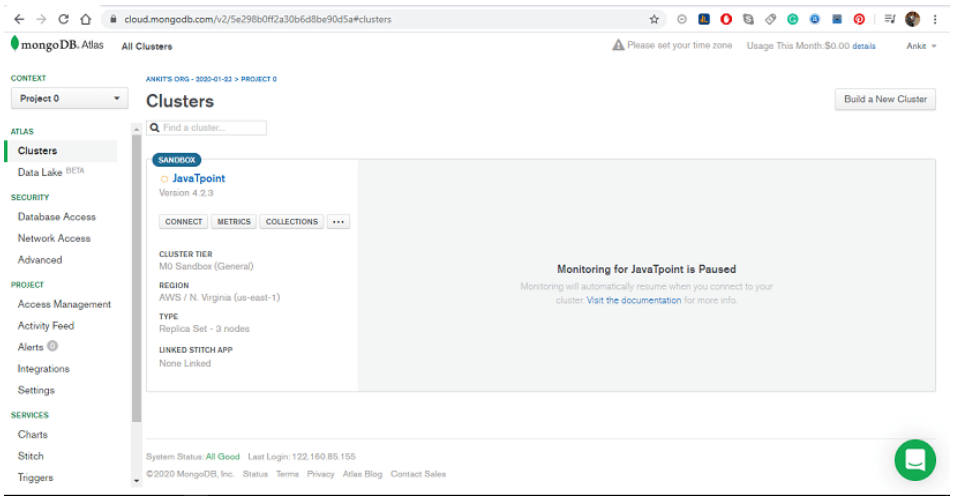
Establishing connection with MongoDB Compass.

There are two methods to connect our deployment in MongoDB compass, either we can use the connection string provided on the MongoDB Atlas or we can fill our deployment information in specified fields.

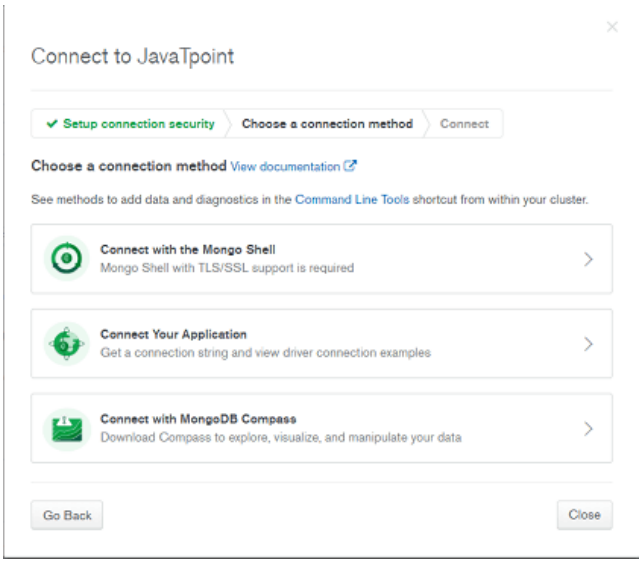
By Pasting Connection String.

Step 1: When you log in to Compass, an initial dialogue will appear.

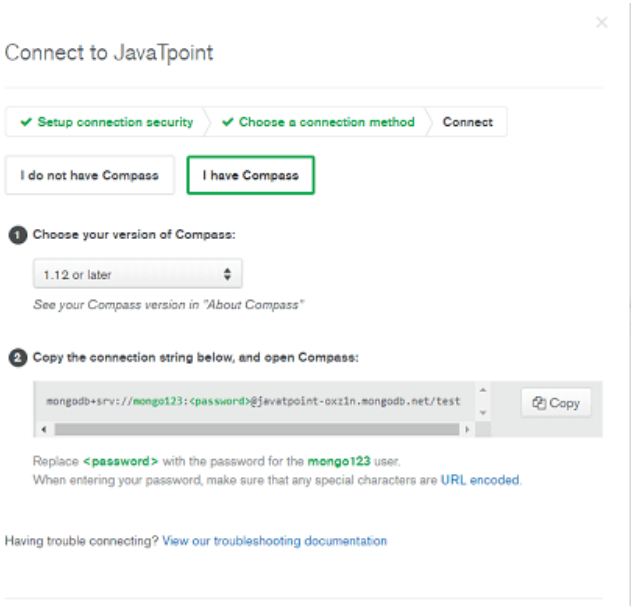
Step 2: To get the deployment connection string for an Atlas cluster, go to your Atlas cluster view.



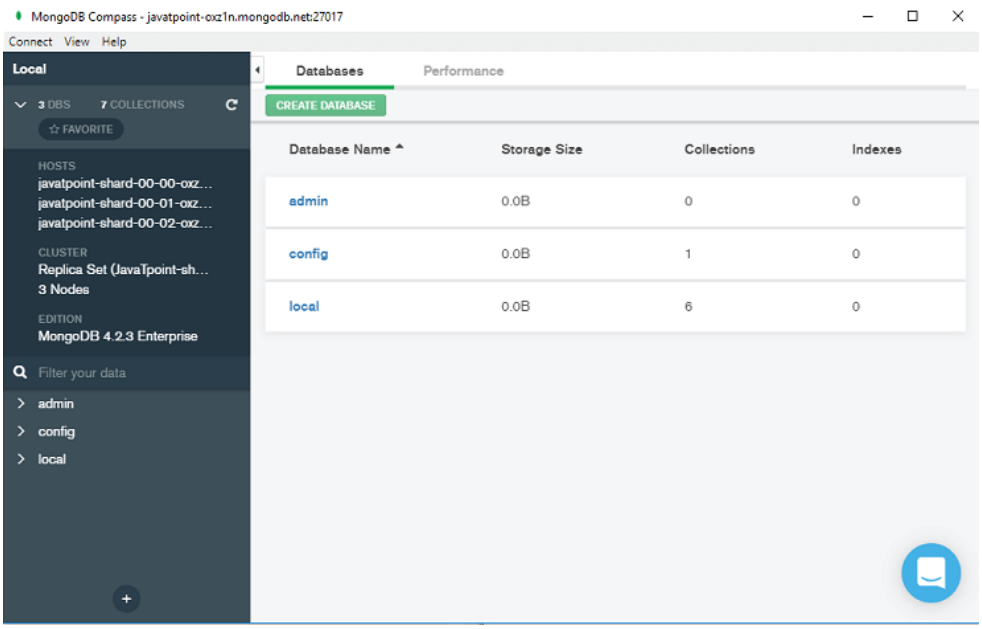
Step 3: Click Connect for the cluster you want to connect.



Step 4: After that, click Connect with MongoDB Compass and copy the provided connection string.



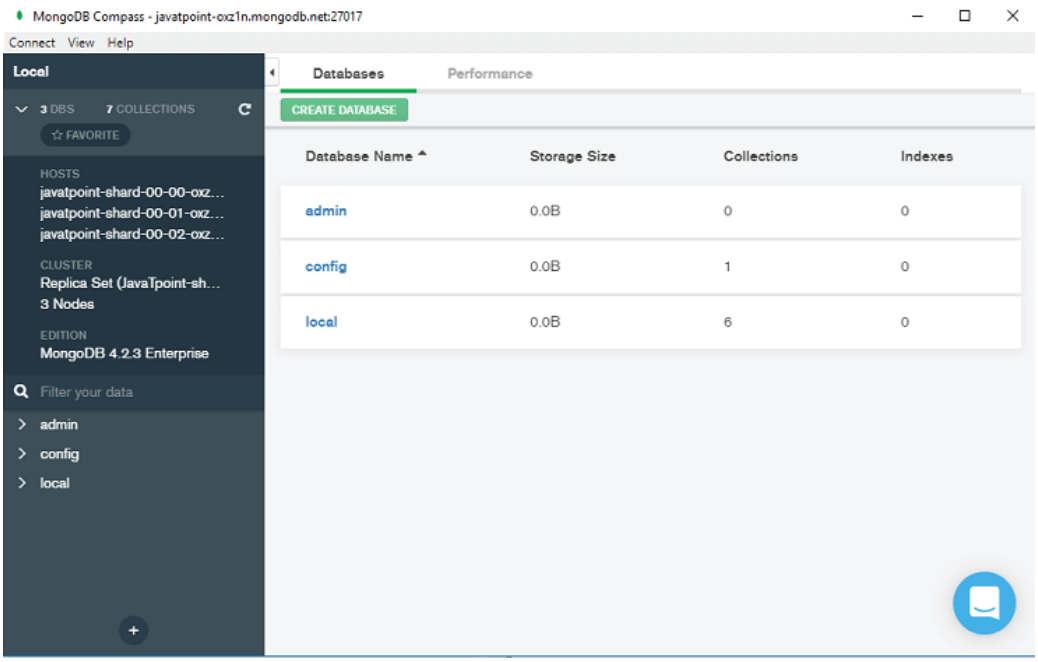
Step 5: Click on connect button, to connect and navigate to the Compass GUI Home Page.



The Compass Home screen displays the details about the MongoDB instance from which Compass is connected, which includes the connection name, the deployment type, hostname, and port, version of MongoDB, performance statistics, and a list of the instance's databases.

Creating and Managing Database using Compass

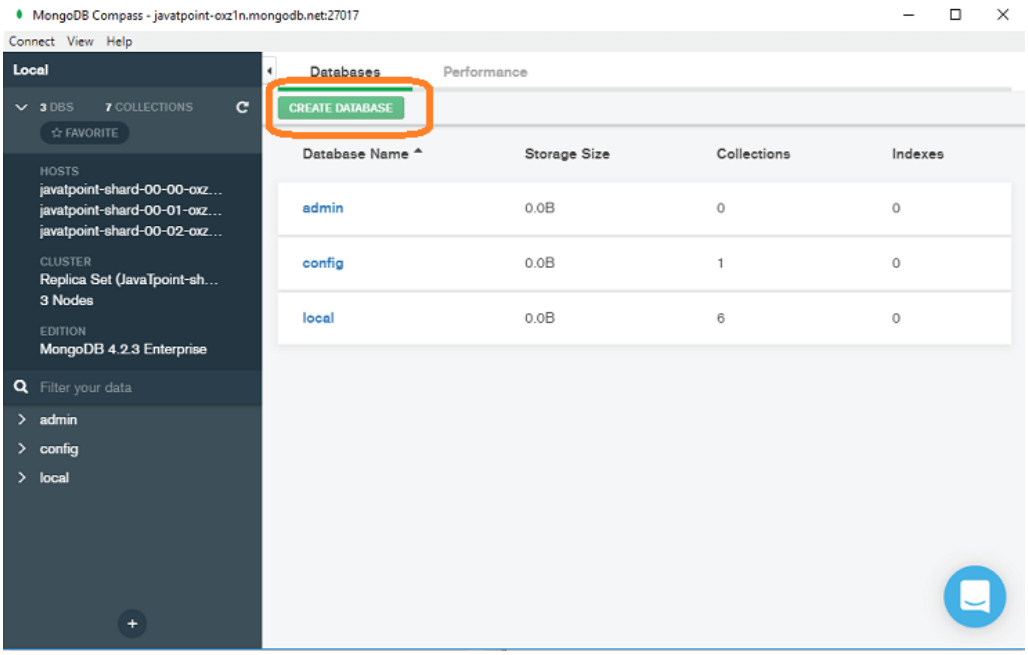
When you are connected to the MongoDB Atlas or Mongo Shell, the following window will appear. Inside this window, you can see the Database tab. The Database window shows the lists of all the existing databases for your MongoDB deployment.



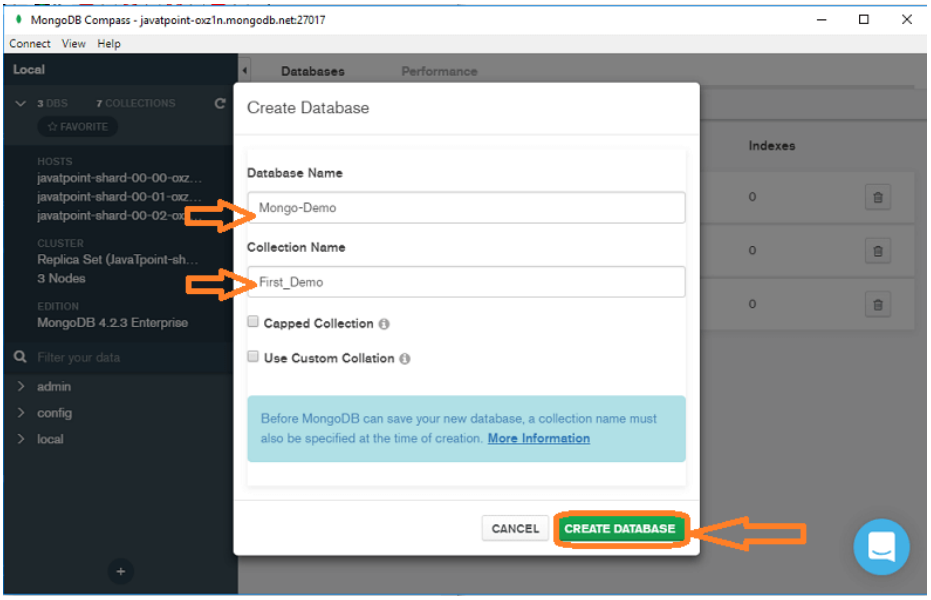
On the above window, when you select a database from the given list to view its collections. You can view database collection when you click the desired Database in the left navigation pan.

Create a Database in Compass

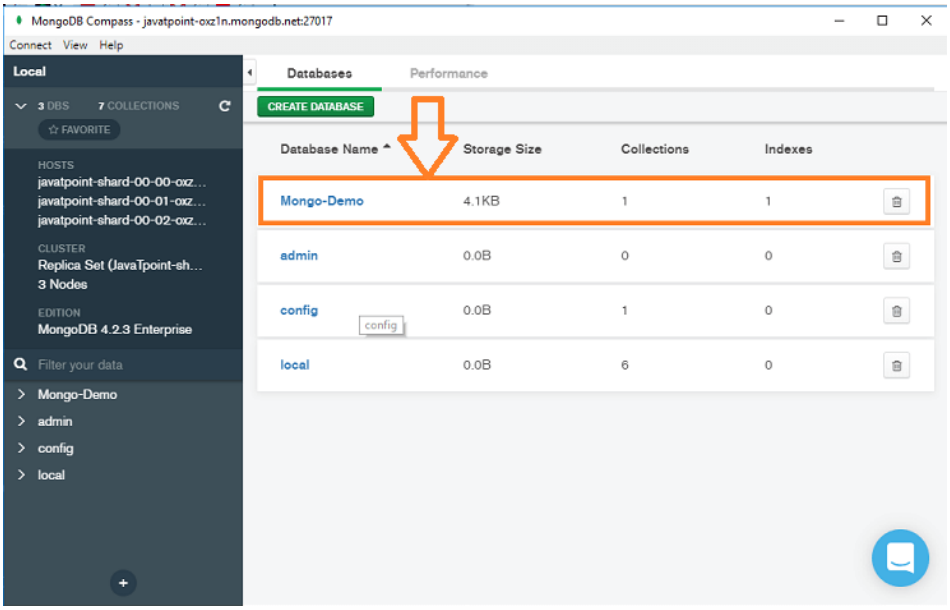
Step 1: Click on the Create Database button from the database tab. It will take you to the **Create Database** pop-up dialogue.



Step 2: In the appeared pop-up window, fill the Database and collection name to create a new database.



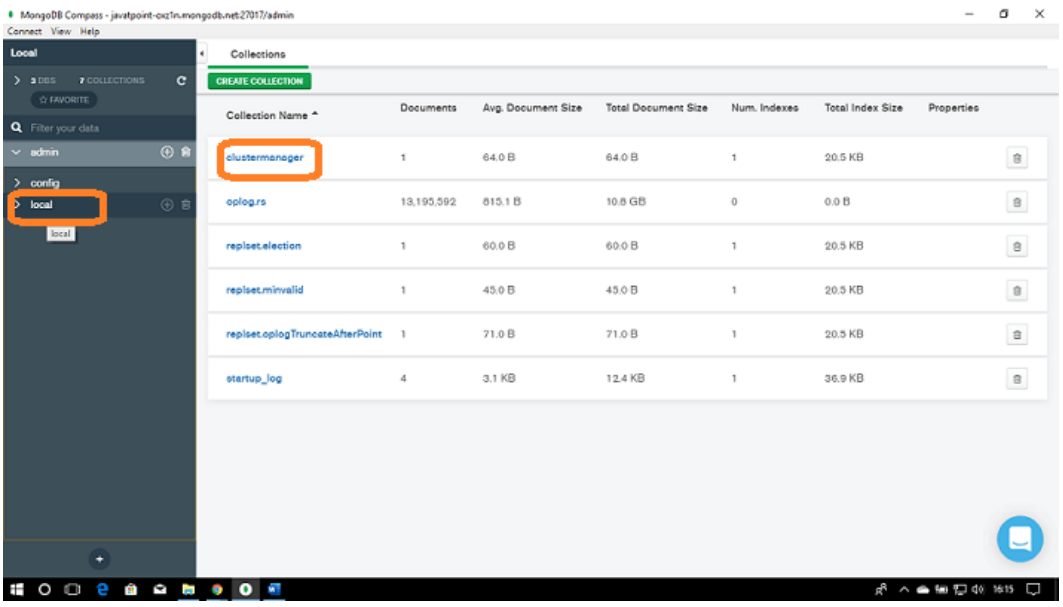
Step 3: Finally, click on ***Create Database*** button to create the Database and collection.



Collections in MongoDB Compass

The Collection window displays the list of all the existing collection and views from the database you have selected. It includes all the name and other related information for the selected collection or view.

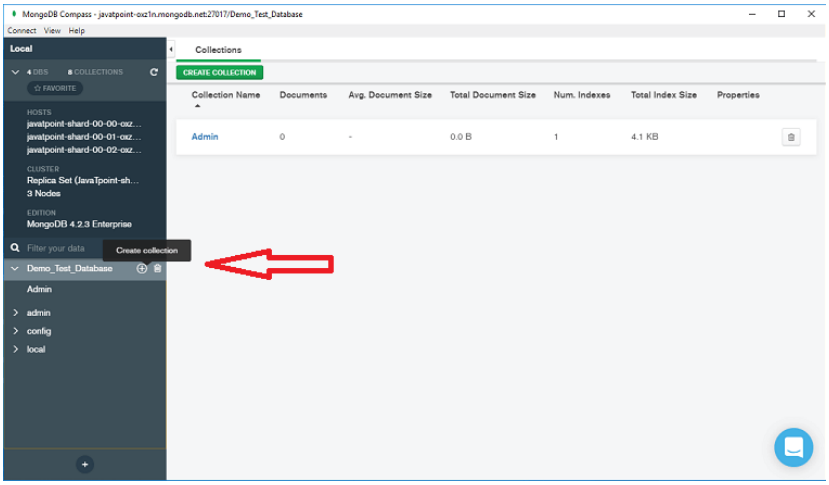
If you want to gain access to the collection of a database, click on the Database Name in the main Database view, or Click on the database in the left navigation pan.



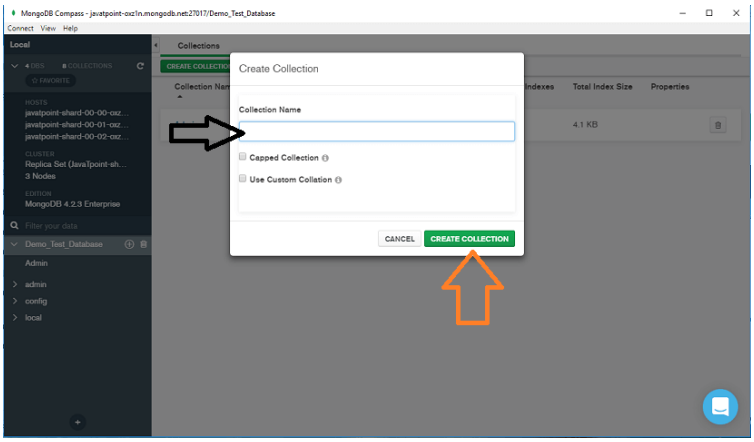
The collection window displays the information like - Collection name, number of documents, size, number of indexes, size of indexes, and Collation properties for the collection.

Create a Collection in MongoDB Compass

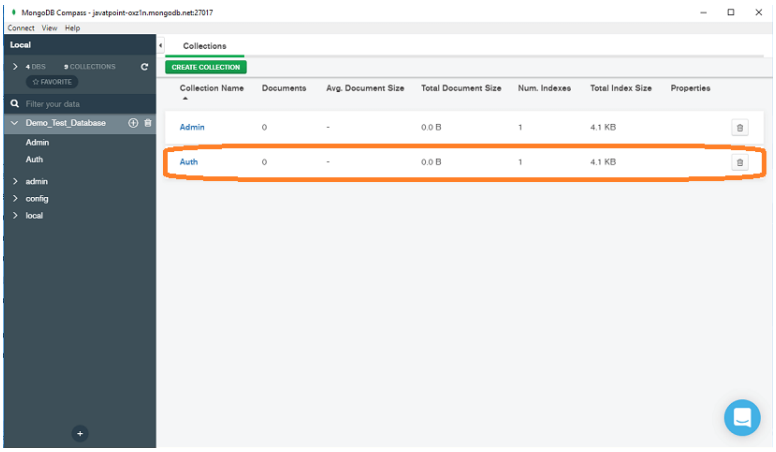
Step 1: Click on the ***Create Collection*** button.



Step 2: After that, fill in the collection details in the ***Create Collection*** dialog.

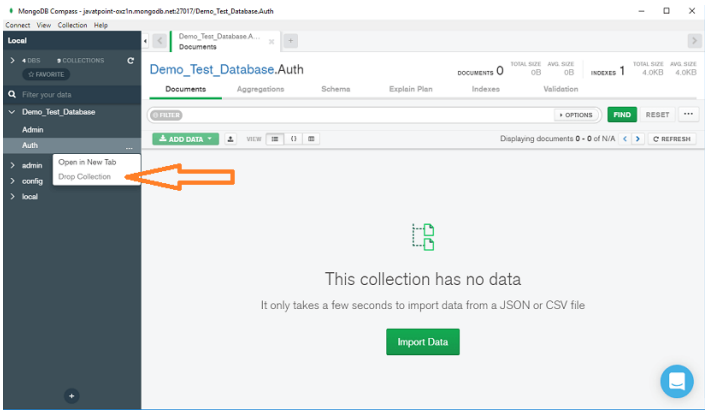


Step 3: Now, click on ***Create Collection*** to create the collection

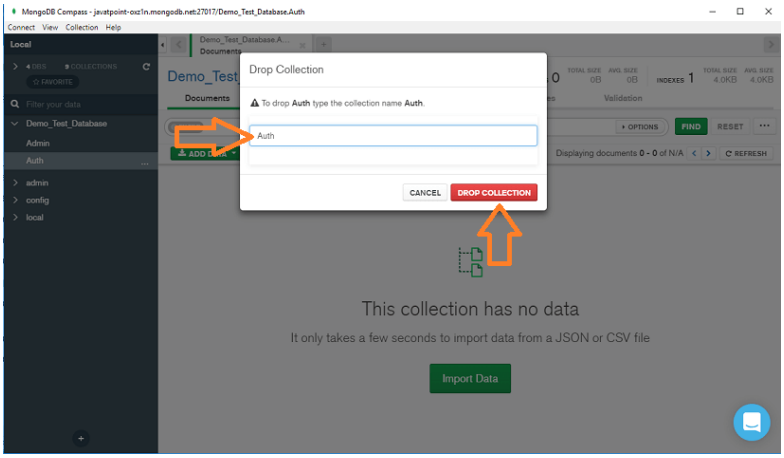


Drop a Collection

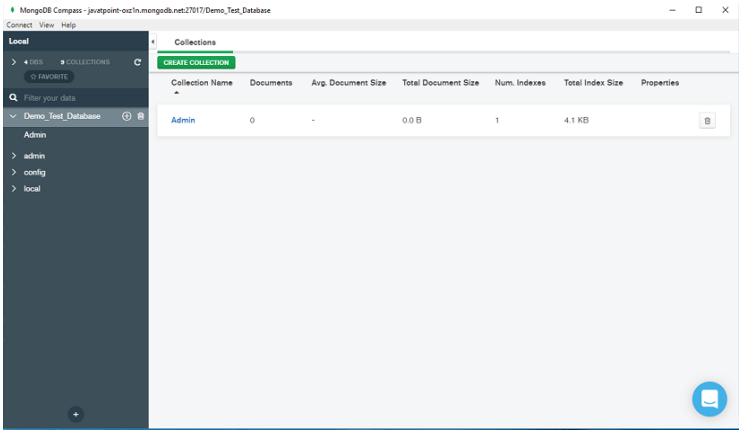
Step 1: In the collection window, click on the bin icon for the collection to delete. When you click on the trash icon, a dialog appears to ask your confirmation.



Step 2: In the appeared pop-up dialogue, enter the name of the collection you want to delete from the database.



Step 3: Finally, click on Drop Collection button to delete the collection.

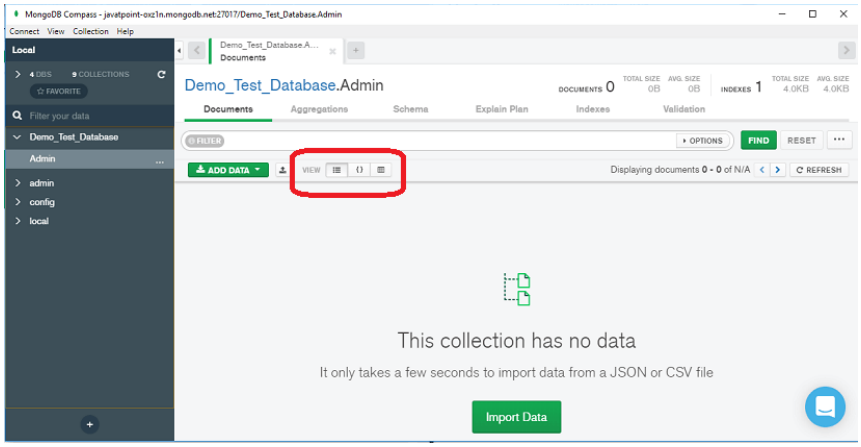


Managing Documents in MongoDB Compass

Documents are the records in a MongoDB collection. Documents are the basic unit of data in MongoDB. Using the document tab, we can perform the following tasks in our selected collection or view:

- **View the documents:** The Document tab provides three ways to access document in MongoDB Compass.
  - **List View** - It is the default view of the Database in the MongoDB Compass. Document will be shown as individual members of the list. In the list view you can easily extend the embedded objects and arrays.
  - **JSON View** - In this view documents will be shown as completely-formatted JSON objects. In this view, MongoDB Compass use extended JSON to display the data-types of fields where the correct data types are used.
  - **Table View** - The table view display documents as a table row. The document fields are shown as a column in the table. When we use table view, we can easily find out the documents that contains specific field values.

  We can use View buttons to select which view we want to use:



- **Insert the document:** We can have two ways to insert documents into our collections:
  - **JSON Mode:** It allows you to write or paste JSON documents in the editor. You can use this mode to insert one or many documents at once as an array in the database.
  - **Field-by-Field editor:** You can create documents in more interactive way using this editor. It allows you to select all the field values and types. It supports only the insertion of a single document at a time.

- **Modify the document:** You can update existing document in your collection. When you make changes to your document, the MongoDB Compass performs a findAndModify operation to update the existing document.
- **Clone the documents:** You can insert new files and documents by cloning (i.e. by making an exact same copy of the document). You can copy the schema and values of an existing document/files in a collection.
- **Delete the documents:** We can delete the document/file depending of the tab whether we are viewing our documents in List, JSON, or Table view.

Data types and operators – T

MongoDB supports many datatypes. Some of them are −

- **String** − This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer** − This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** − This type is used to store a boolean (true/ false) value.
- **Double** − This type is used to store floating point values.
- **Min/ Max keys** − This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** − This type is used to store arrays or list or multiple values into one key.
- **Timestamp** − ctimestamp. This can be handy for recording when a document has been modified or added.
- **Object** − This datatype is used for embedded documents.
- **Null** − This type is used to store a Null value.
- **Symbol** − This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date** − This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** − This datatype is used to store the document's ID.
- **Binary data** − This datatype is used to store binary data.
- **Code** − This datatype is used to store JavaScript code into the document.
- **Regular expression** − This datatype is used to store regular expression.

MongoDB Query and Projection Operator

The MongoDB query operator includes comparison, logical, element, evaluation, Geospatial, array, bitwise, and comment operators.

MongoDB Comparison Operators

$eq

The $eq specifies the equality condition. It matches documents where the value of a field equals the specified value.

Syntax:

```
{<field>: {$eq: <value>} }
```

Example:

```
db.books.find ( { price: { $eq: 300 } } )
```

The above example queries the books collection to select all documents where the value of the price filed equals 300.

$gt

The $gt chooses a document where the value of the field is greater than the specified value.

Syntax:

```
{ field: { $gt: value } }
```

Example:

```
db.books.find ( { price: { $gt: 200 } } )
```

$gte

The $gte choose the documents where the field value is greater than or equal to a specified value.

Syntax:

```
{ field: { $gte: value} }
```

Example:

```
db.books.find ( { price: { $gte: 250 } } )
```

$in

The $in operator choose the documents where the value of a field equals any value in the specified array.

Syntax:

```
{ filed: { $in: [ <value1>, <value2>, ……] } }
```

Example:

```
db.books.find( { price: { $in: [100, 200] } } )
```

$lt

The $lt operator chooses the documents where the value of the field is less than the specified value.

Syntax:

```
{ field: { $lt: value } }
```

Example:

```
db.books.find ( { price: { $lt: 20 } } )
```

$lte

The $lte operator chooses the documents where the field value is less than or equal to a specified value.

Syntax:

```
{ field: { $lte: value } }
```

Example:

```
db.books.find ( { price: { $lte: 250 } } )
```

$ne

The $ne operator chooses the documents where the field value is not equal to the specified value.

Syntax:

```
{ <field>: { $ne: <value> } }
```

Example:

```
db.books.find ( { price: { $ne: 500 } } )
```

$nin

The $nin operator chooses the documents where the field value is not in the specified array or does not exist.

Syntax:

```
{ field : { $nin: [ <value1>, <value2>, .... ] } }
```

Example:

```
db.books.find ( { price: { $nin: [ 50, 150, 200 ] } } )
```

MongoDB Logical Operator

$and

The $and operator works as a logical AND operation on an array. The array should be of one or more expressions and chooses the documents that satisfy all the expressions in the array.

Syntax:

```
{ $and: [ { <exp1> }, { <exp2> }, ....]}
```

Example:

```
db.books.find ( { $and: [ { price: { $ne: 500 } }, { price: { $exists: true } } ] } )
```

$not

The $not operator works as a logical NOT on the specified expression and chooses the documents that are not related to the expression.

Syntax:

```
{ field: { $not: { <operator-expression> } } }
```

Example:

```
db.books.find ( { price: { $not: { $gt: 200 } } } )
```

$nor

The $nor operator works as logical NOR on an array of one or more query expression and chooses the documents that fail all the query expression in the array.

Syntax:

```
{ $nor: [ { <expression1> } , { <expresion2> } , ..... ] }
```

Example:

```
db.books.find ( { $nor: [ { price: 200 }, { sale: true } ] } )
```

$or

It works as a logical OR operation on an array of two or more expressions and chooses documents that meet the expectation at least one of the expressions.

Syntax:

```
{ $or: [ { <exp_1> }, { <exp_2> }, ... , { <exp_n> } ] }
```

Example:

```
db.books.find ( { $or: [ { quantity: { $lt: 200 } }, { price: 500 } ] } )
```

MongoDB Element Operator

$exists

The exists operator matches the documents that contain the field when Boolean is true. It also matches the document where the field value is null.

Syntax:

```
{ field: { $exists: <boolean> } }
```

Example:

```
db.books.find ( { qty: { $exists: true, $nin: [ 5, 15 ] } } )
```

$type

The type operator chooses documents where the value of the field is an instance of the specified BSON type.

Syntax:

```
{ field: { $type: <BSON type> } }
```

Example:

```
db.books.find ( { "bookid" : { $type : 2 } } );
```

MongoDB Evaluation Operator

$expr

The expr operator allows the use of aggregation expressions within the query language.

Syntax:

```
{ $expr: { <expression> } }
```

Example:

```
db.store.find( { $expr: {$gt: [ "$product" , "price" ] } } )
```

$jsonSchema

It matches the documents that satisfy the specified JSON Schema.

Syntax:

```
{ $jsonSchema: <JSON schema object> }
```

$mod

The mod operator selects the document where the value of a field is divided by a divisor has the specified remainder.

Syntax:

```
{ field: { $mod: [ divisor, remainder ] } }
```

Example:

```
db.books.find ( { qty: { $mod: [ 200, 0] } } )
```

$regex

It provides regular expression abilities for pattern matching strings in queries. The MongoDB uses regular expressions that are compatible with Perl.

Syntax:

```
{ <field>: /pattern/<options> }
```

Example:

```
db.books.find( { price: { $regex: /789$/ } } )
```

$text

The $text operator searches a text on the content of the field, indexed with a text index.

Syntax:

```
{
 $text:
  {
   $search: <string>,
   $language: <string>,
   $caseSensitive: <boolean>,
   $diacriticSensitive: <boolean>
  }
}
```

Example:

```
db.books.find( { $text: { $search: "Othelo" } } )
```

$where

The "where" operator is used for passing either a string containing a JavaScript expression or a full JavaScript function to the query system.

Example:

```
db.books.find( { $where: function() {
  return (hex_md5(this.name)== "9b53e667f30cd329dca1ec9e6a8")
} } );
```

MongoDB Geospatial Operator

$geoIntersects

It selects only those documents whose geospatial data intersects with the given GeoJSON object.

Syntax:

```
{
 <location field>: {
   $geoIntersects: {
```

```
        $geometry: {
          type: "<object type>" ,
          coordinates: [ <coordinates> ]
        }
      }
    }
}
```

Example:

```
        db.places.find(
{
  loc: {
    $geoIntersects: {
      $geometry: {
        type: "Triangle" ,
        coordinates: [
          [ [ 0, 0 ], [ 3, 6 ], [ 6, 1 ] ]
        ]
      }
    }
  }
}
```

$geoWithin

The geoWithin operator chooses the document with geospatial data that exists entirely within a specified shape.

Syntax:

```
        {
<location field>: {
  $geoWithin: {
    $geometry: {
      type: <"Triangle" or "Rectangle"> ,
      coordinates: [ <coordinates> ]
    }
```

```
  }
}
```

$near

The near operator defines a point for which a geospatial query returns the documents from close to far.

Syntax:

```
        {
<location field>: {
 $near: {
  $geometry: {
    type: "Point" ,
    coordinates: [ <longitude> , <latitude> ]
  },
  $maxDistance: <distance in meters>,
  $minDistance: <distance in meters>
 }
}
```

Example:

```
          db.places.find(
{
 location:
  { $near :
    {
     $geometry: { type: "Point",  coordinates: [ -73.9667, 40.78 ] },
     $minDistance: 1000,
     $maxDistance: 5000
    }
  }
}
```

$nearSphere

The nearsphere operator specifies a point for which the geospatial query returns the document from nearest to farthest.

Syntax:

```
{
 $nearSphere: [ <x>, <y> ],
 $minDistance: <distance in radians>,
 $maxDistance: <distance in radians>
}
```

Example:

```
db.legacyPlaces.find(
  { location : { $nearSphere : [ -73.9667, 40.78 ], $maxDistance: 0.10 } }
)
```

$all

It chooses the document where the value of a field is an array that contains all the specified elements.

Syntax:

```
{ <field>: { $all: [ <value1> , <value2> ... ] } }
```

Example:

```
db.books.find( { tags: { $all: [ "Java", "MongoDB", "RDBMS" ] } } )
```

$elemMatch

The operator relates documents that contain an array field with at least one element that matches with all the given query criteria.

Syntax:

```
{ <field>: { $elemMatch: { <query1>, <query2>, ... } } }
```

Example:

```
db.books.find(
  { results: { $elemMatch: { $gte: 500, $lt: 400 } } }
)
```

$size

It selects any array with the number of the element specified by the argument.

Syntax:

```
db.collection.find( { field: { $size: 2 } } );
```

MongoDB Bitwise Operator

$bitsAllClear

It matches the documents where all the bit positions given by the query are clear infield.

Syntax:

```
{ <field>: { $bitsAllClear: <numeric bitmask> } }
```

Example:

```
db.inventory.find( { a: { $bitsAllClear: [ 1, 5 ] } } )
```

$bitsAllSet

The bitallset operator matches the documents where all the bit positions given by the query are set in the field.

Syntax:

```
{ <field>: { $bitsAllSet: <numeric bitmask> } }
```

Example:

```
db.inventory.find( { a: { $bitsAllClear: [ 1, 5 ] } } )
```

$bitsAnyClear

The bitAnyClear operator matches the document where any bit of positions given by the query is clear in the field.

Syntax:

```
{ <field>: { $bitsAnyClear: <numeric bitmask> } }
```

Example:

```
db.inventory.find( { a: { $bitsAnyClear: [ 5, 10 ] } } )
```

$bitsAnySet

It matches the document where any of the bit positions given by the query are set in the field.

Syntax:

```
{ <field>: { $bitsAnySet: <numeric bitmask> } }
```

Example:

```
db.inventory.find( { a: { $bitsAnySet: [ 1, 5 ] } } )
```

MongoDB comments operator

$comment

The $comment operator associates a comment to any expression taking a query predicate.

Syntax:

```
db.inventory.find( { <query>, $comment: <comment> } )
```

Example:

```
        db.inventory.find(
{
  x: { $mod: [ 1, 0 ] },
  $comment: "Find Odd values."
}
```

MongoDB Projection Operator

$

The $ operator limits the contents of an array from the query results to contain only the first element matching the query document.

Syntax:

```
db.books.find( { <array>: <value> ... },
   { "<array>.$": 1 } )
db.books.find( { <array.field>: <value> ...},
   { "<array>.$": 1 } )
```

$elemMatch

The content of the array field made limited using this operator from the query result to contain only the first element matching the element $elemMatch condition.

Syntax:

```
db.library.find( { bookcode: "63109" },
{ students: { $elemMatch: { roll: 102 } } } )
```

$meta

The meta operator returns the result for each matching document where the metadata associated with the query.

Syntax:

```
{ $meta: <metaDataKeyword> }
```

Example:

```
db.books.find(
<query>,
{ score: { $meta: "textScore" } }
```

$slice

It controls the number of values in an array that a query returns.

Syntax:

```
db.books.find( { field: value }, { array: {$slice: count } } );
```

Example:

```
db.books.find( {}, { comments: { $slice: [ 200, 100 ] } } )
```