MONGO DB

MongoDB is a popular **open-source**, **NoSQL** database that stores data in a **document-oriented format**. Unlike traditional relational databases, which store data in tables, MongoDB stores data as JSON-like documents, making it more flexible and scalable. It was developed by MongoDB Inc. and first released in 2009.

MongoDB was created to address the need for a database system that could handle the growing volume of unstructured data being generated by modern web and mobile applications. It was designed to be a high-performance, horizontally scalable database system that could be used for a wide range of applications.

The name **"MongoDB"** is derived from the word **"humongous,"** which reflects the scalability and flexibility of the database.

Since its release, MongoDB has become one of the most popular NoSQL databases in the world, with a large and active community of developers and users. It is used by companies of all sizes, from startups to large enterprises, and is particularly well-suited for applications that require high performance, scalability, and flexibility.

MongoDB has also evolved over time to include a wide range of features and capabilities, including advanced querying, indexing, and aggregation capabilities, as well as support for geospatial data, full-text search, and graph processing. It also provides enterprise-level security and monitoring features, making it suitable for use in mission-critical applications.

MongoDB is used in a wide variety of applications and industries, including:

* **Web and mobile applications:** MongoDB is often used as the primary database for web and mobile applications, where it provides high performance and scalability, as well as support for flexible data models.
* **E-commerce:** MongoDB is used by many e-commerce sites to store product catalogs, customer data, and order information, as well as to provide real-time analytics and personalized recommendations.
* **Social networking:** MongoDB is used by social networking sites to store user profiles, activity feeds, and social graphs, as well as to provide real-time analytics and recommendations.
* **Gaming:** MongoDB is used in the gaming industry to store user data, game progress, and other game-related information, as well as to provide real-time analytics and recommendations.
* **Financial services:** MongoDB is used in the financial services industry to store and analyze large volumes of data, such as transaction data, customer data, and market data.
* **Healthcare:** MongoDB is used in the healthcare industry to store and manage patient data, medical records, and other healthcare-related information, as well as to provide real-time analytics and insights.
* **Government:** MongoDB is used by many government agencies to store and manage large volumes of data, such as census data, weather data, and traffic data, as well as to provide real-time analytics and insights.

Overall, MongoDB is a versatile and flexible database system that can be used in a wide range of applications and industries, where it provides high performance, scalability, and flexibility.

### **A Brief History of MongoDB**

#### History of MongoDb

* MongoDB is a **NoSQL database** that was developed by **10gen**, a company founded by **Dwight Merriman and Eliot Horowitz** in 2007.
* The **first version** of MongoDB was released in **2009**, and it quickly gained popularity among developers due to its ease of use, **scalability**, and **flexibility**.
* In 2013, **10gen changed its name to MongoDB Inc**. to better reflect its focus on the development of the MongoDB database.
* In 2017, MongoDB Inc. went public, and the company has continued to grow and expand its offerings, including the introduction of a **cloud-based database service called MongoDB Atlas**.
* Today, MongoDB is used by a wide range of companies and organizations,

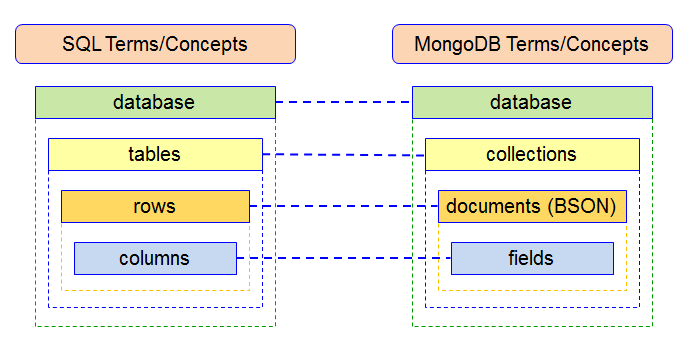
The name "MongoDB" is derived from the word "humongous," reflecting the database's ability to store and manage large amounts of data.

MongoDB was created by Dwight Merriman and Eliot Horowitz, who were both developers at DoubleClick, which was later acquired by Google. While working on a project to manage large amounts of data across multiple data centers, they found that traditional relational databases didn't provide the performance and scalability they needed. They decided to build their own database system, which eventually became MongoDB.

MongoDB quickly gained popularity among developers due to its flexibility, scalability, and ease of use. It also has a wide range of features, including support for indexing, replication, sharding, and aggregation.

Today, MongoDB is used by some of the world's largest companies, including Adobe, eBay, Forbes, and The New York Times, to manage and process their data. The company MongoDB Inc. has also developed a range of related products and services, including cloud hosting, analytics, and consulting services.

### **How MongoDB Works**



MongoDB is a document-oriented, NoSQL database that stores data in a JSON-like format, which is often referred to as BSON (Binary JSON). Unlike traditional relational databases, MongoDB does not use a fixed schema or table structure, but rather stores data in flexible and dynamic documents.

Here's how MongoDB works based on the following points:

#### Database :

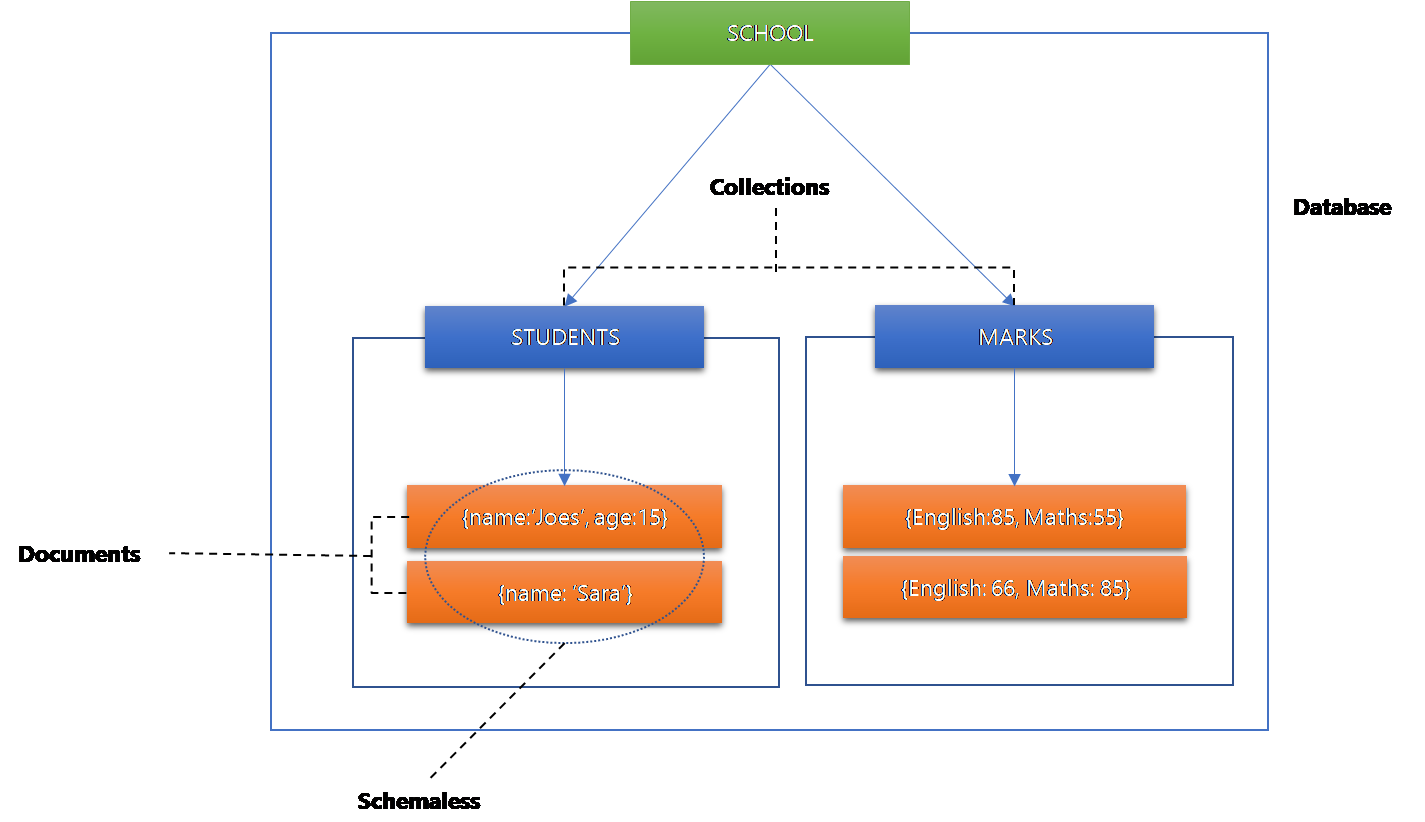
In MongoDB, a database is a container for collections. It is where all the data is stored. You can think of a database as a namespace for collections. You can create as many databases as you need, and each database can have one or more collections.

#### Collections :

A collection in MongoDB is a group of related documents. It is similar to a table in a relational database, but without a fixed schema. You can add or remove fields to a collection at any time, without affecting other documents in the collection. A collection can have one or more indexes, which can be used to optimize query performance.

#### Documents :

A document in MongoDB is a JSON-like data structure that represents a single instance of data. It is similar to a row in a relational database, but with a more flexible and dynamic structure. A document can contain any number of fields, which can be of different data types, and can have nested structures.



Documents in a collection do not have to follow a fixed schema, allowing for more flexibility in data modeling. For example, one document in a collection could have additional fields compared to other documents. This schemaless approach is one of the key features of MongoDB.

To summarize, MongoDB stores data in a JSON-like format, with a flexible and dynamic structure, without a fixed schema. Data is organized into databases, which contain collections of related documents. This document-oriented approach offers greater flexibility and scalability than traditional relational databases, making it popular for web and mobile applications that require high performance and scalability.

|  |  |  |
| --- | --- | --- |
| **Description** | **MySQL** | **MongoDB** |
| Data Storage | Tables | Collections |
|  |  |  |
| Data Record | Row | Document |
| Field | Column | Field |
| Index | Index | Index |
| Primary Key | Primary Key | Primary Key (automatically created for the \_id field) |
| Query Language | Structured Query Language (SQL) | MongoDB Query Language (MQL) |
| Joining Tables | JOIN clause | Embedded documents or manual references |
| Transactions | ACID-compliant transactions | Multi-document transactions (in recent versions) |
| Replication | Master-slave replication | Replica sets |
| Sharding | Partitioning based on a sharding key | Sharding based on a sharding key or hashed shard key |
| Data Modeling | ER modeling | No fixed schema, document-oriented data modeling |
| Use Cases | Traditional data-driven applications | Web applications, real-time analytics, and high-volume transactional systems |

### **Understanding the JSON Data Format**

In MongoDB, JSON is the primary data format for storing and working with data. MongoDB stores data in a binary-encoded format called BSON (Binary JSON), which is a superset of JSON that includes additional data types and features.

MongoDB uses JSON-like documents to represent data in the database. These documents are stored in collections, which are similar to tables in a traditional relational database. Each document is a self-contained unit that represents a single instance of data, and can contain any number of fields, with each field consisting of a key-value pair.

{

"name":"Joes",

"age":35,

"gender":"male",

"married":**true**,

"address":{

"street":"cherry Road",

"city":"Salem",

"state":"Tamil Nadu"

},

"hobbies":[

{

"name":"Cooking"

},

{

"name":"Sports"

}

]

}

The JSON-like format of MongoDB documents allows for flexible data modeling, since documents in a collection do not have to follow a fixed schema. This makes MongoDB well-suited for use cases where the structure of data can change frequently, such as in web and mobile applications.

In MongoDB, JSON is used not only for data storage, but also for data querying and manipulation. MongoDB provides a rich set of query operators and aggregation functions that allow you to work with and analyze JSON-like documents in powerful and flexible ways. Overall, JSON is an essential part of working with MongoDB, as it is the primary format for representing and working with data in the database.

### **Exploring the Thriving MongoDB Ecosystem**

The MongoDB ecosystem is a collection of tools, services, and technologies that extend the capabilities of MongoDB and help developers and businesses build and manage applications more easily. Here are some key components of the MongoDB ecosystem:

* **MongoDB Server:** The core component of the ecosystem is the MongoDB database server, which stores data in a flexible, document-based format.
* **MongoDB Atlas:** A fully managed cloud database service that provides automatic scaling, backup, and monitoring for MongoDB deployments in the cloud.
* **MongoDB Compass:** A visual tool for exploring and managing MongoDB databases, with features such as real-time performance monitoring, query profiling, and schema visualization.
* **MongoDB Charts:** A data visualization tool that enables developers and analysts to create charts and graphs from MongoDB data without writing any code.
* **MongoDB Stitch:** A serverless platform that enables developers to build and deploy microservices that integrate with MongoDB data and other cloud services.
* **MongoDB Mobile:** A mobile database that enables developers to build offline-first applications that sync with MongoDB Atlas in the cloud.
* **MongoDB Connector for BI:** A connector that enables business intelligence (BI) tools such as Tableau, Qlik, and Excel to connect to MongoDB data and create reports and dashboards.
* **MongoDB Community:** A community of users, contributors, and advocates for MongoDB, who provide support, contribute to open-source projects, and share knowledge and best practices.
* **MongoDB University:** An online learning platform that offers courses, certifications, and tutorials on MongoDB, covering topics such as database administration, data modeling, and application development.

Overall, the MongoDB ecosystem offers a rich set of tools and services that enable developers and businesses to build, manage, and analyze data in powerful and flexible ways.

### **The Powerful Features of MongoDB**

**Aggregation** is a powerful feature in MongoDB that allows you to process and analyze data across multiple documents in a collection. It enables you to perform complex data manipulations, transformations, and calculations on large datasets

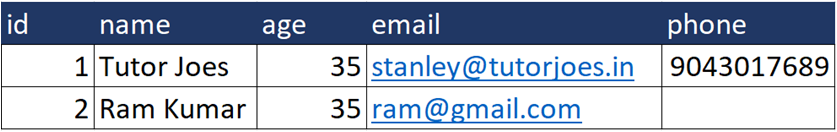


MongoDB is a **schema-less database**, which means that it does not require a pre-defined schema or structure for the data. Unlike traditional relational databases, where data is stored in tables with pre-defined columns and relationships between tables, MongoDB stores data in flexible and dynamic documents.

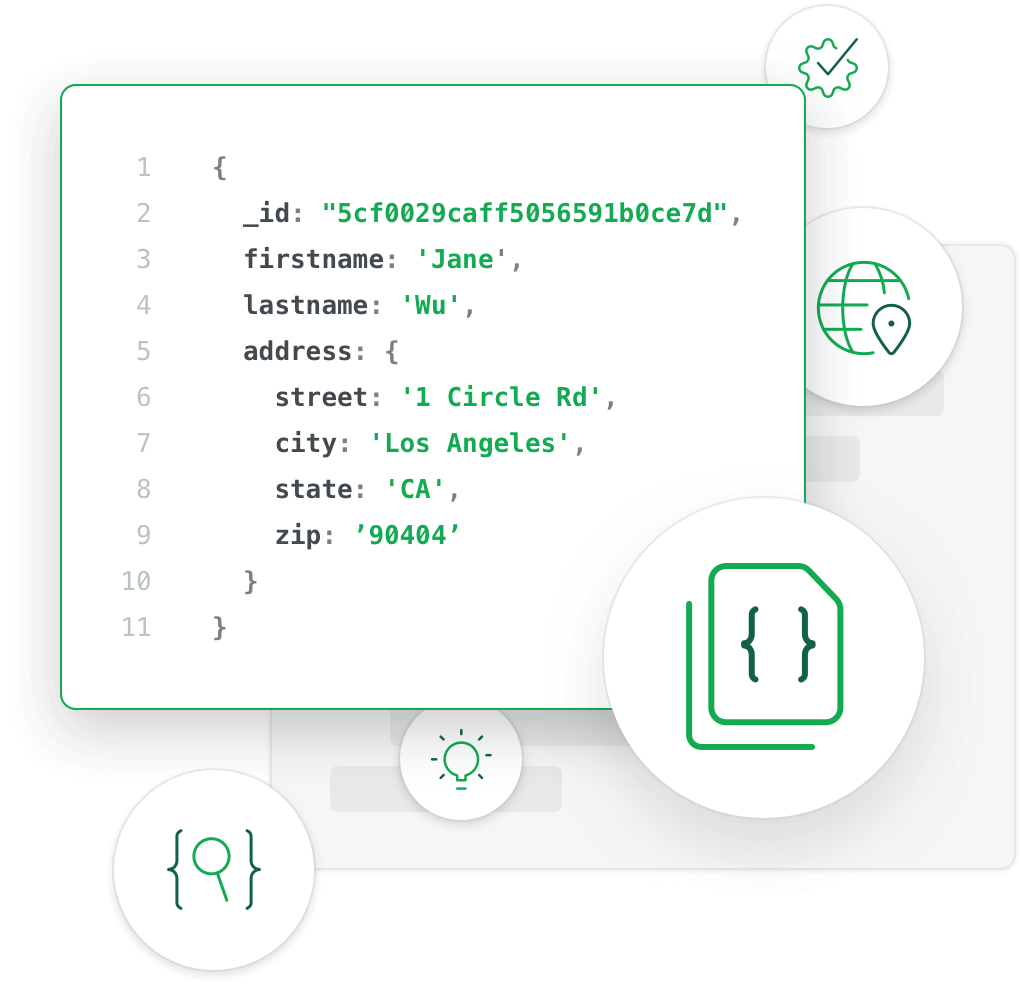
#### MongoDB



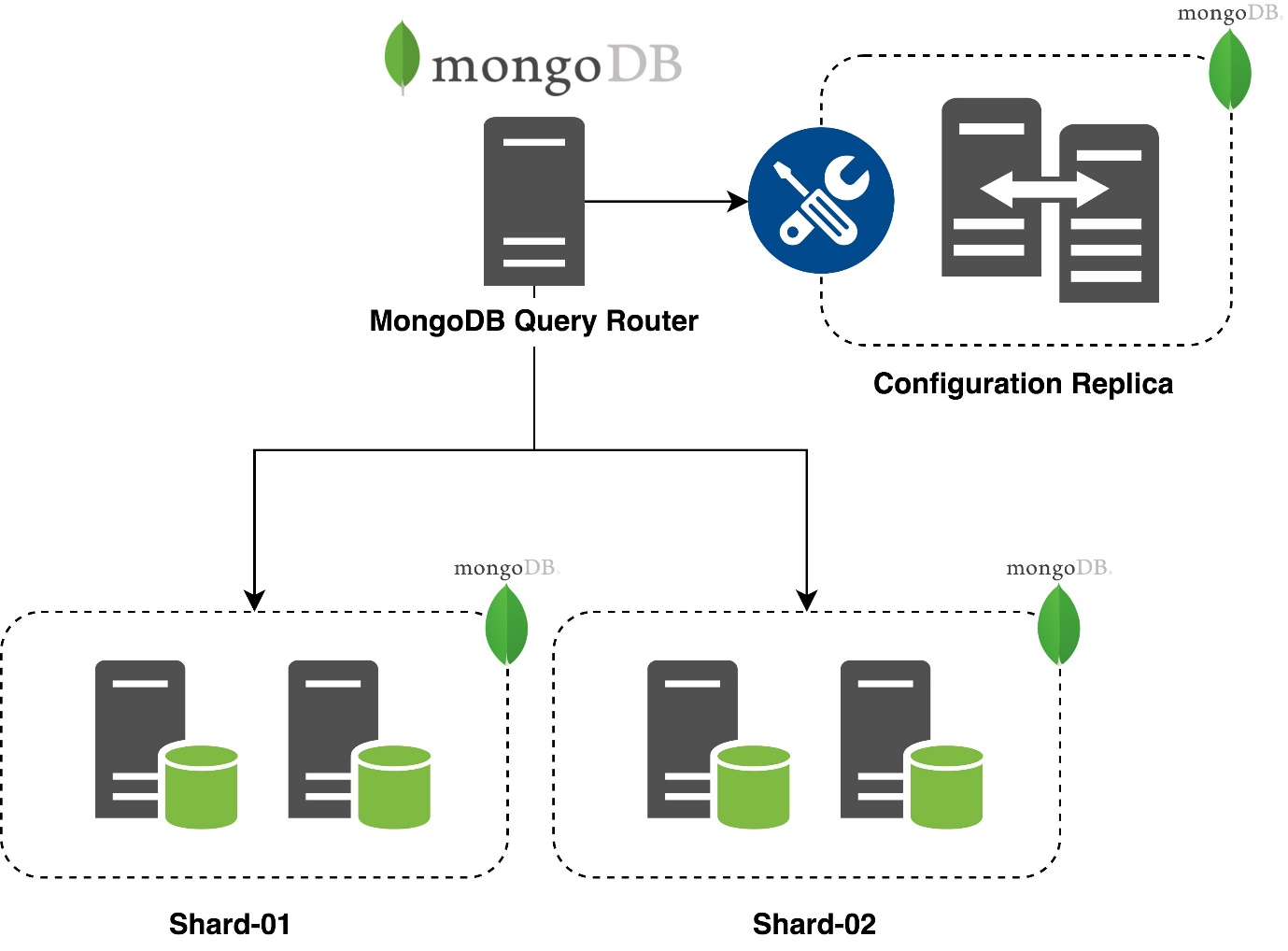
#### SQL Schema



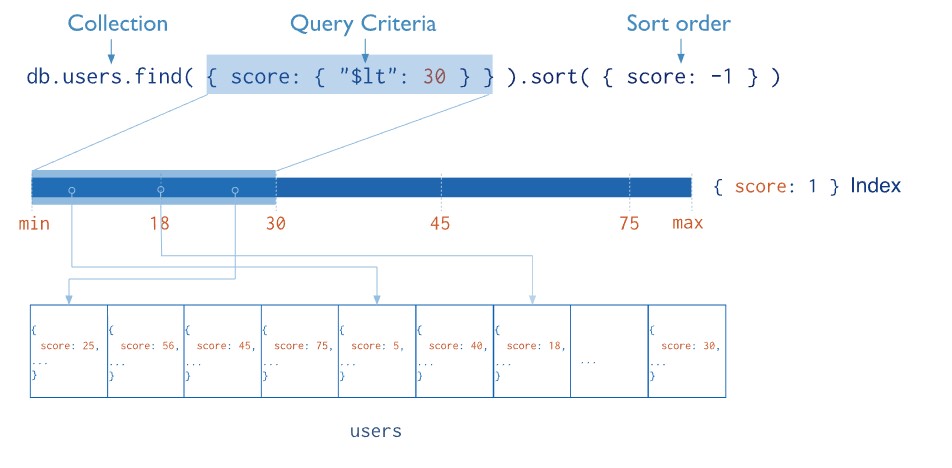
MongoDB is **document-oriented** this means that data is stored in the form of documents, which are similar to JSON (JavaScript Object Notation) objects, instead of being stored in tables with fixed columns and rows like in a traditional relational database. Each document has unique system generated key.



**Sharding** allows MongoDB to handle large datasets and high traffic loads while maintaining high performance and scalability. Sharding is a type of database portioning that separates very large databases into smaller faster more easily managed part called data shards.

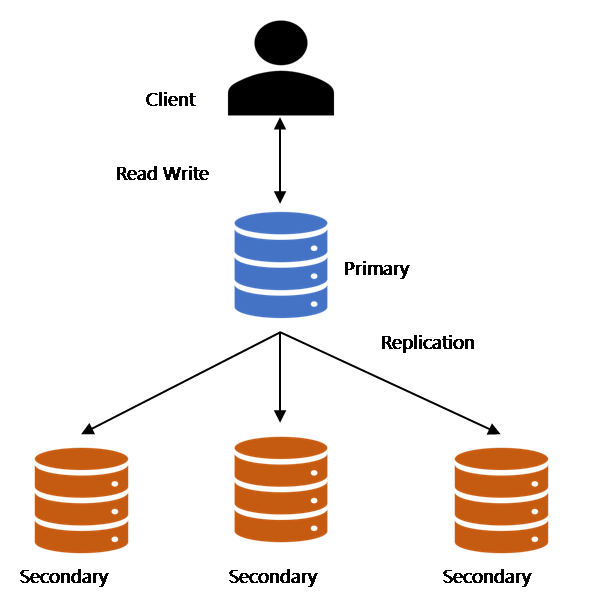


**Indexing** is one of the important option to improve the search performances.so we should index those fields which are matched with our searching criteria.



**High Performance** MongoDB shows high availability and scability.it has better query response for indexing and replication.

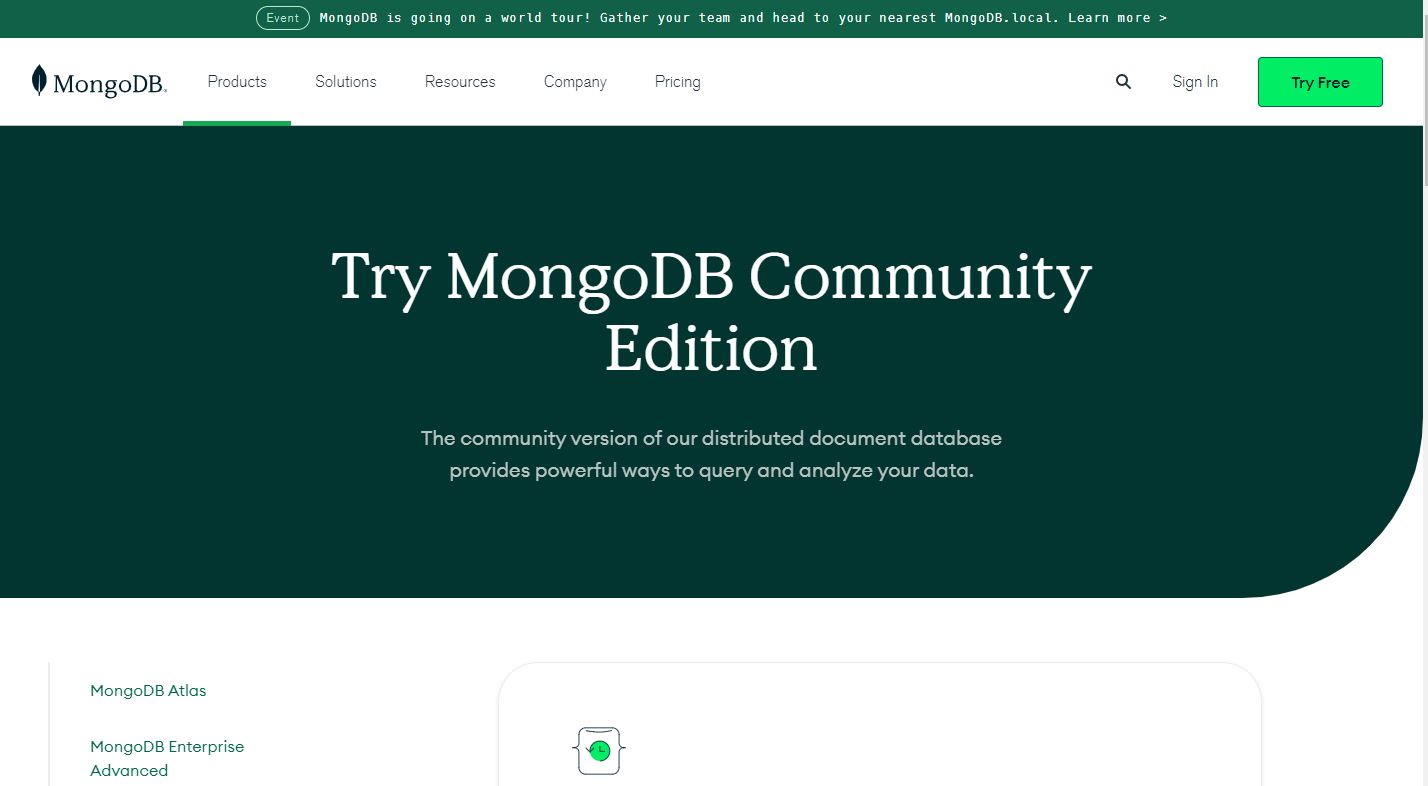
**Replication:** A replica set in MongoDB consists of one primary node and one or more secondary nodes, which replicate the data from the primary node. The primary node is responsible for handling write operations, while the secondary nodes replicate the data from the primary node and can handle read operations. If the primary node fails, one of the secondary nodes is elected as the new primary node



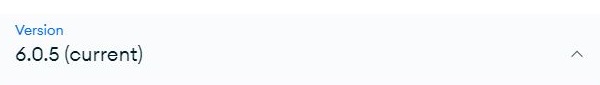
### 

### **Step-by-Step Guide to MongoDB Installation on Windows**

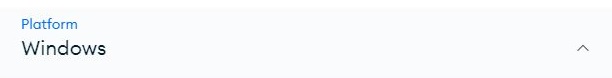
**Step 1 :** Go to the MongoDB download page at [**(https://www.mongodb.com/try/download/community/)**](https://www.mongodb.com/try/download/community/) and select the **"Windows"** tab.



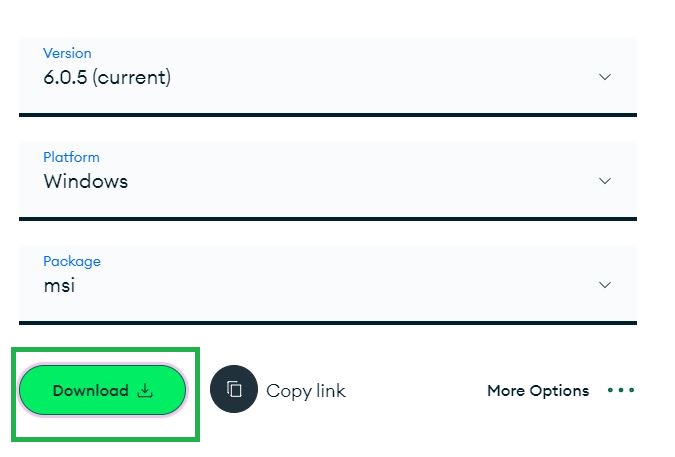
**Step 2 :** Under **"Version"**, select the desired version of MongoDB to download.



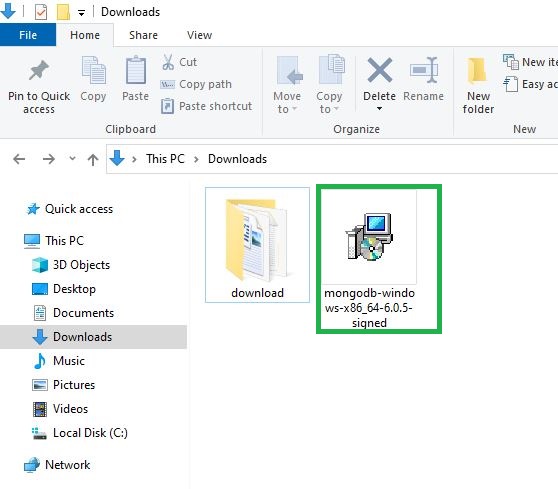
**Step 3 :** Under **"OS"**, select the appropriate version of Windows.



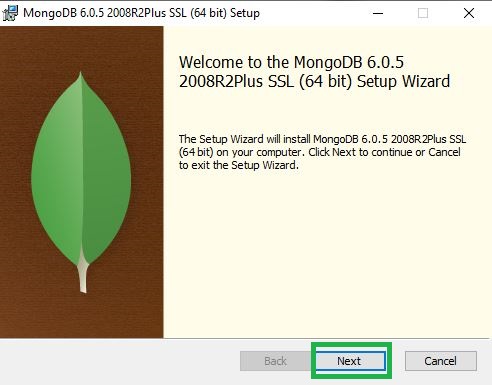
**Step 4 :** Click the **"Download"** button to download the MongoDB installer.



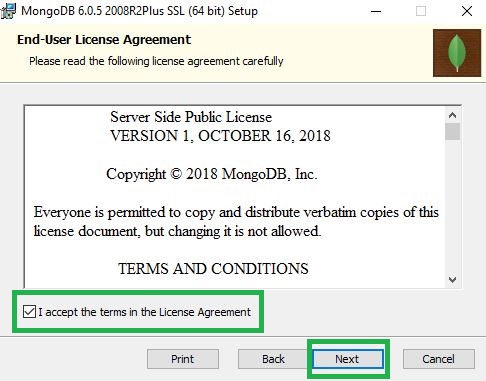
**Step 5 :** **Install MongoDB** : **Double-click** on the downloaded file to begin the installation process. Follow the instructions in the installer to complete the installation.



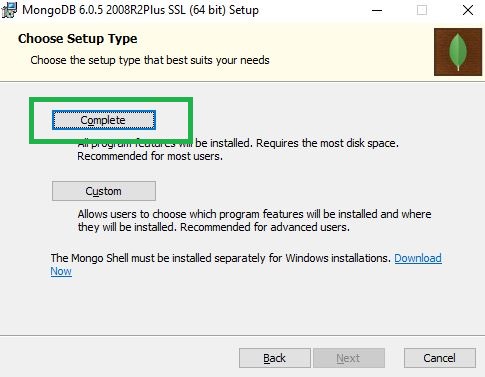
**Step 6 :** Click **"Next"** to start install MongoDB.



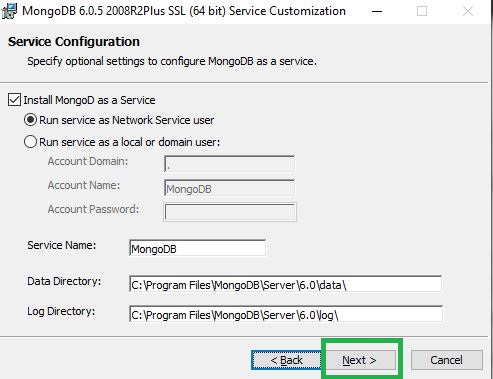
**Step 7 :** Select the **"I accept the terms in the License Agreement"** option and click **"Install."**



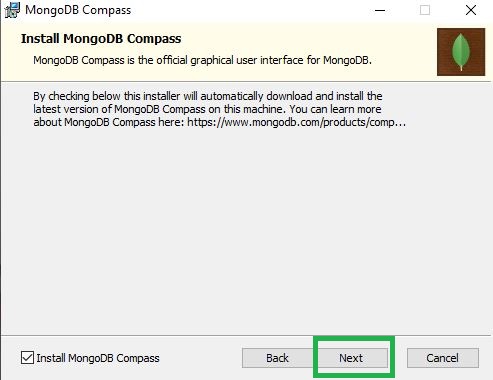
**Step 8 :** Choose the setup type that best suits your needs . Click the **"Complete"** button.



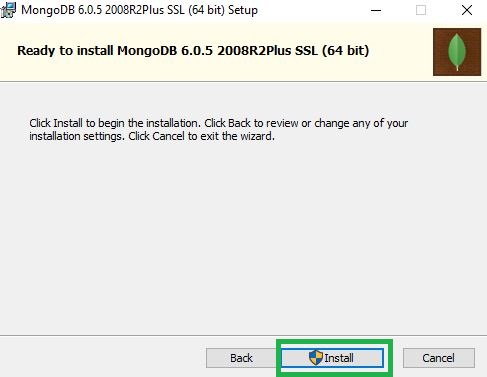
**Step 9 :** Click the **"Next"** button.



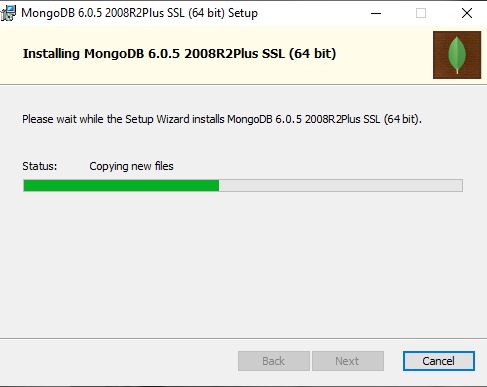
**Step 10 :** Select the **"Install MongoDB Compass"** option and click **"Next."**



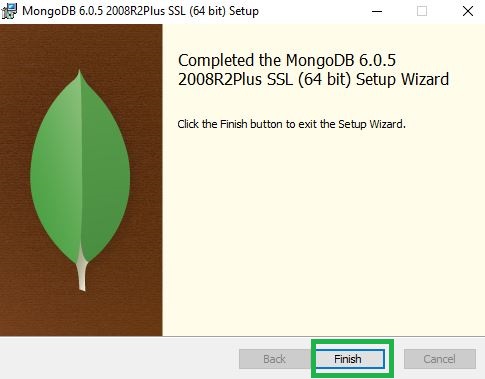
**Step 11 :** Click the **"Install"** button.



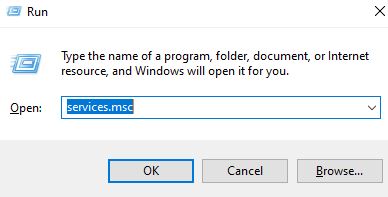
Wait for the installation to complete. This may take several minutes.

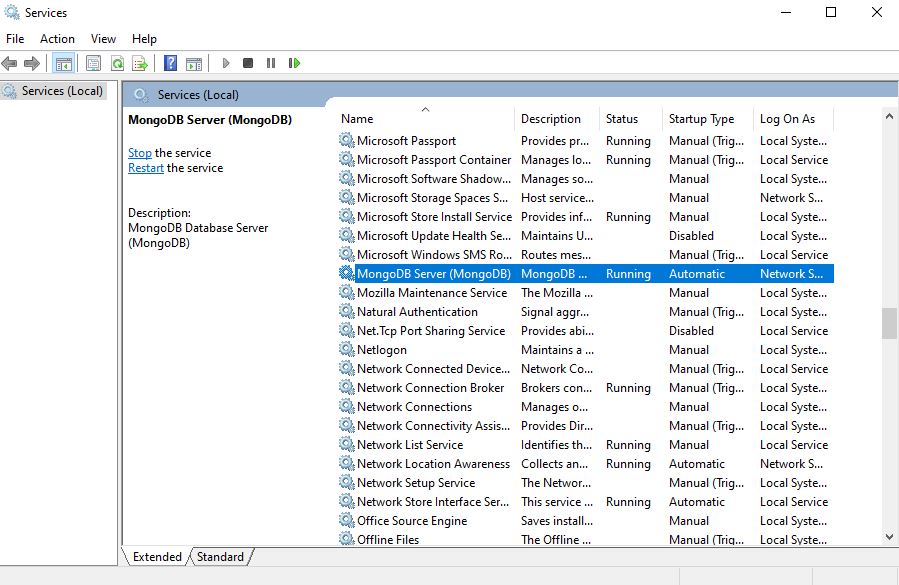


**Step 12 :** Click the **"Finish"** button.



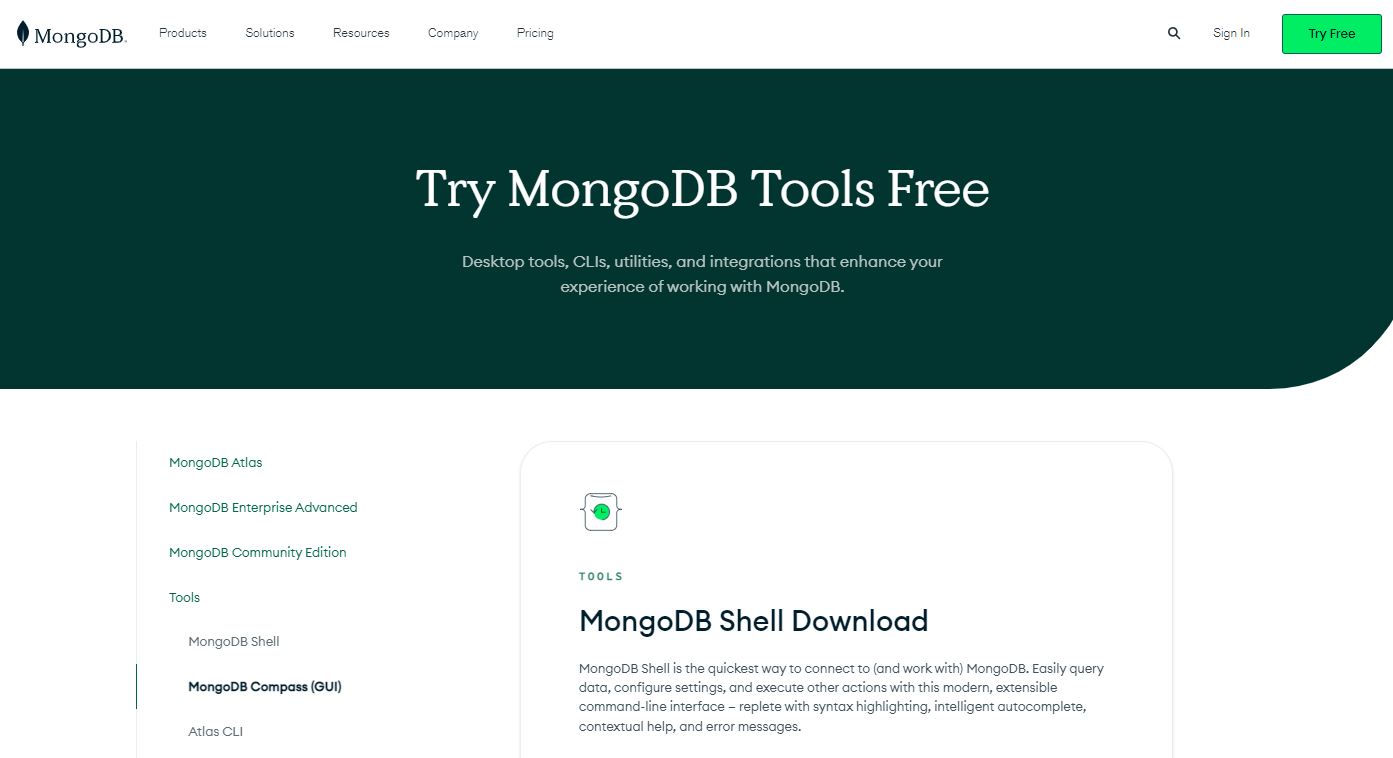
**Step 13 :** After the installation is complete, open a command prompt by pressing the Windows **key + R** and typing **"services.msc"** in the Run dialog box. Press Enter to open the command prompt.



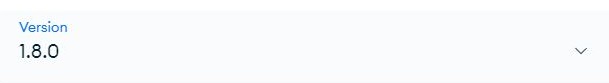


### **Step-by-Step Guide to MongoDB Shell Installation**

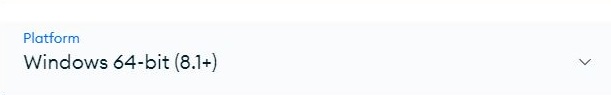
**Step 1 :** Go to the MongoDB download page at [**(https://www.mongodb.com/try/download/shell/)**](https://www.mongodb.com/try/download/shell/) and select the **"Windows"** tab.



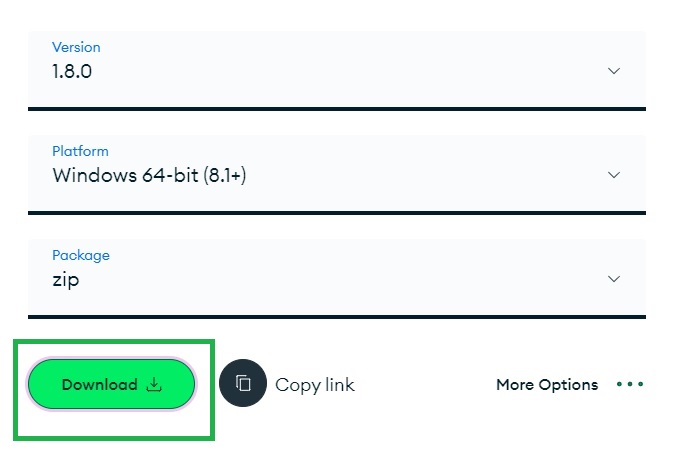
**Step 2 :** Under **"Version"**, select the desired version of MongoDB to download.



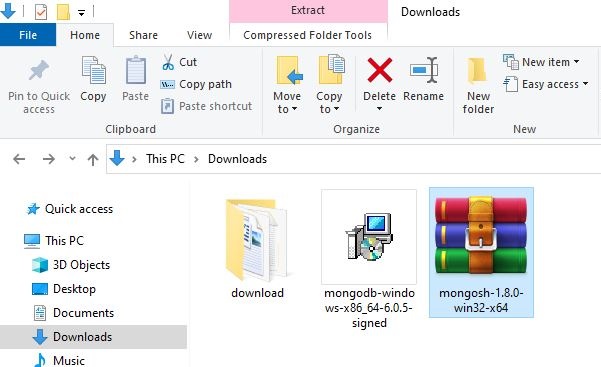
**Step 3 :** Under **"OS"**, select the appropriate version of Windows.



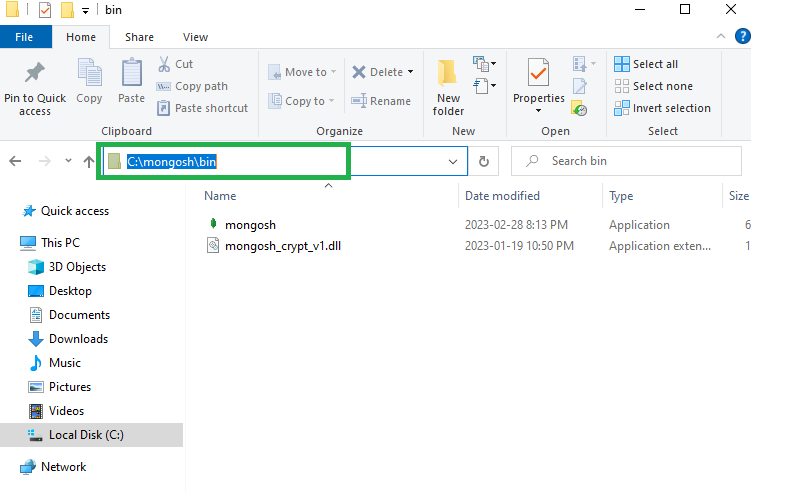
**Step 4 :** Click the **"Download"** button to download the MongoDB installer for Windows.



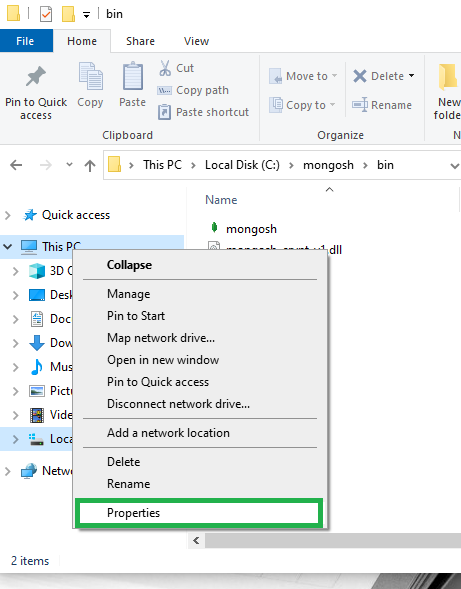
**Step 5 :** Go to download folder zip file on your computer. It should have a **.zip** file extension. **Right-click** on the zip file and select **"Extract All"** or **"Extract Here"** from the context menu. Click on the **"Extract"** or **"Next"** button to start the extraction process. Wait for the extraction to complete. The time it takes to extract the zip file depends on the size of the file and the performance of your computer



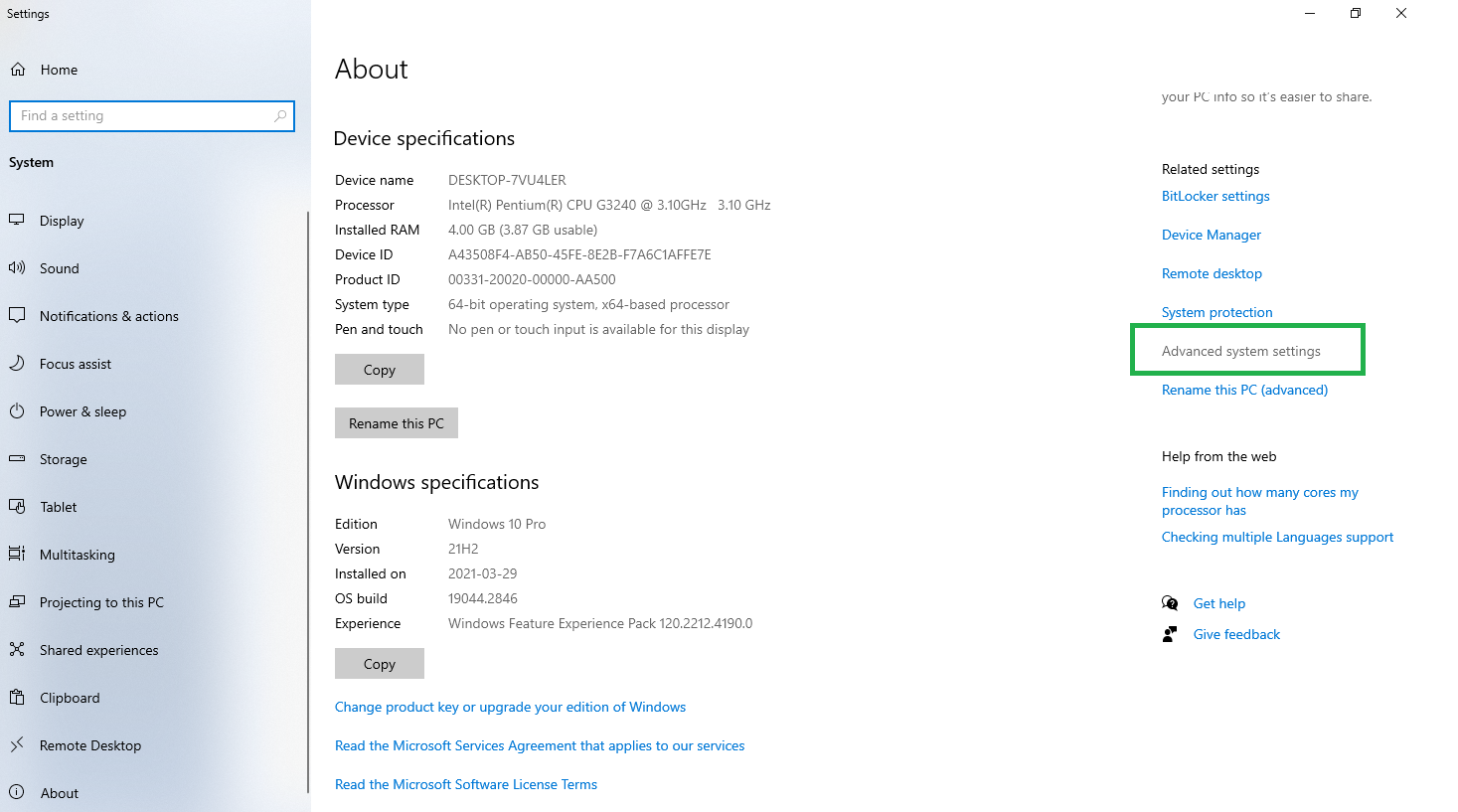
**Step 6 :** Copy **"bin"** installed directory path.



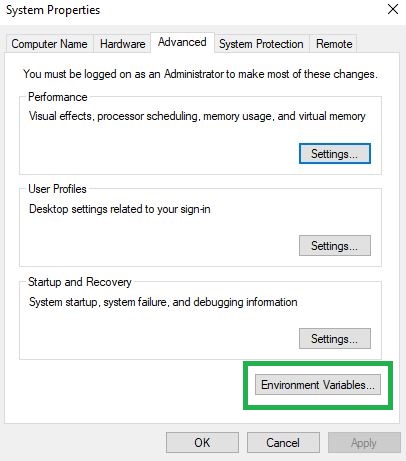
**Step 7 :** Go to **This PC** -> **Right Click** -> **Properties**.



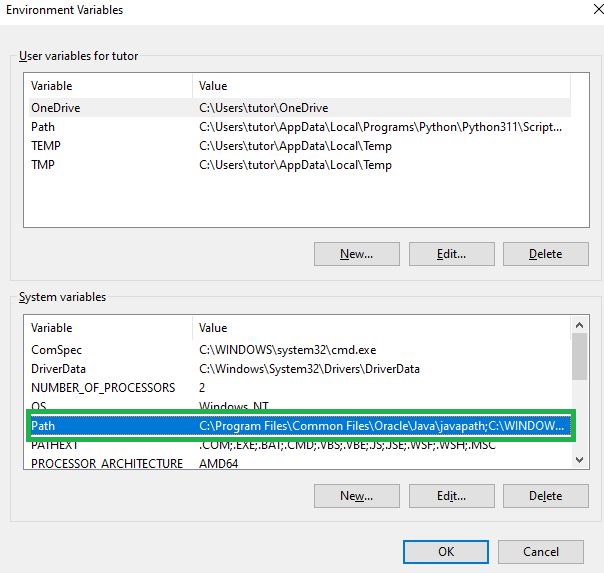
**Step 8 :** Click **"Advanced system settings"** on the left Side.



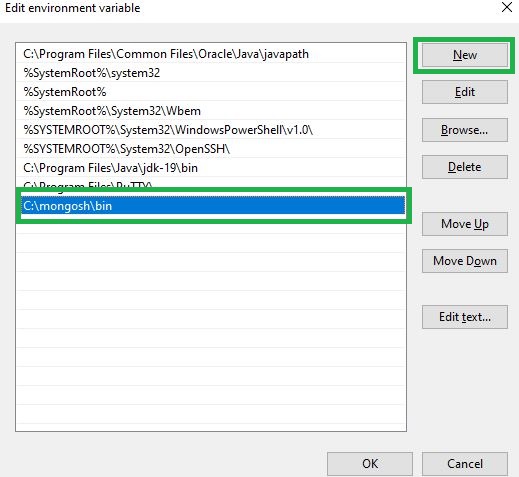
**Step 9 :** Click **"Environment variables Button"**.



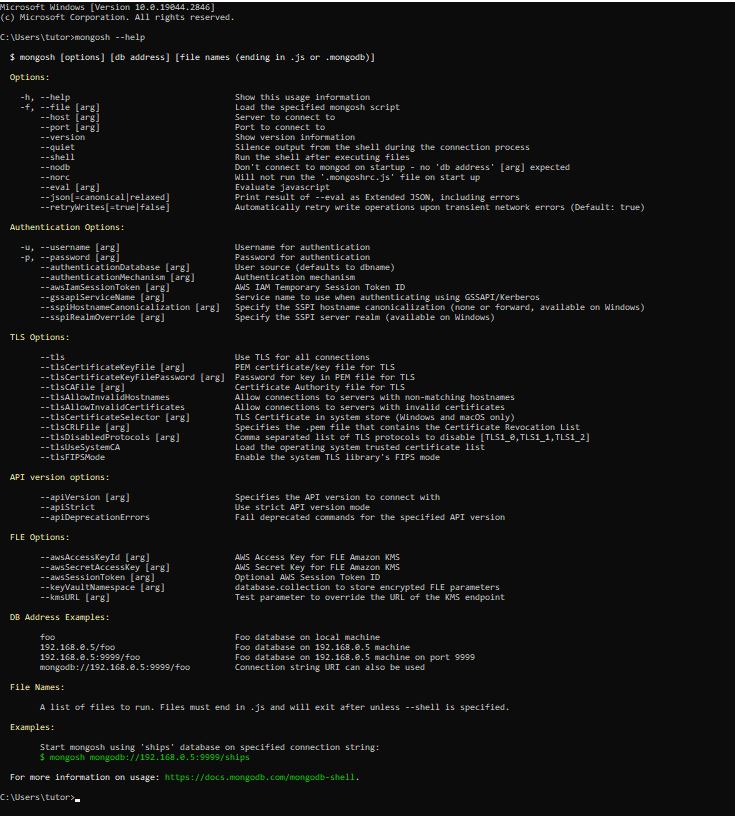
**Step 10 :** Select the **system variable** -> **Path** -> **Click New**



**Step 11 :** You will see a TABLE listing for all existing PATH entries. **Click** -> **"New"** -> **"paste"** your path -> then click **"ok"**



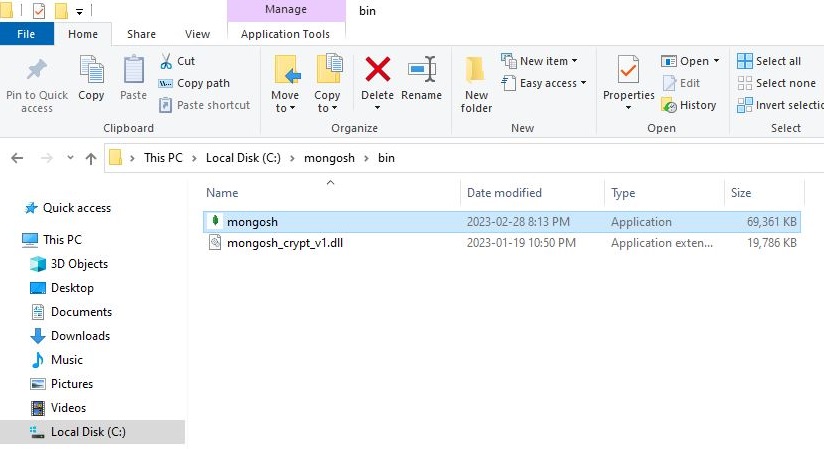
**Step 12 :** Start -> Enter **cmd (or) Command Prompt** -> Choose Command Prompt -> Type **mongosh --help** and press Enter



### **Basic Select and Insert Queries**

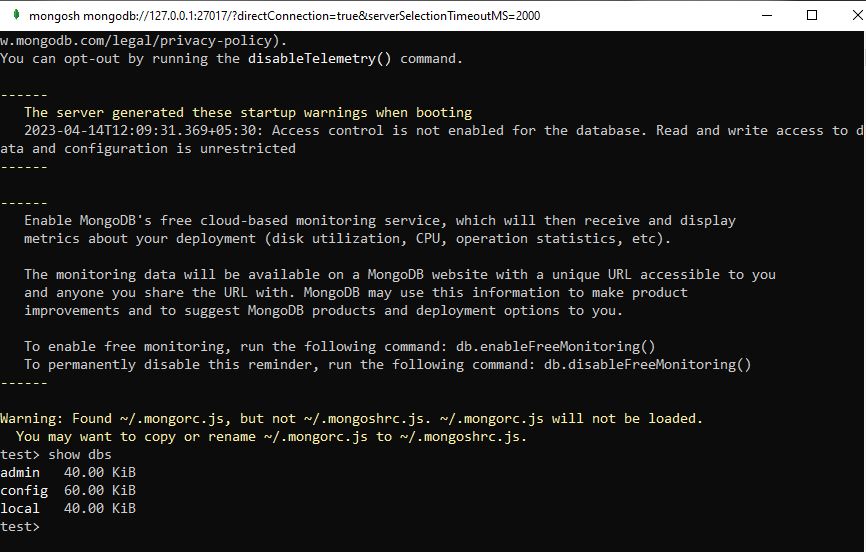
#### OPEN MONGOSH

**Double-click** on the **mongosh.exe**



#### VIEW ALL DATABASES

you can use the **show** command with the dbs option to display the list of databases. Type **"show dbs"** and press Enter

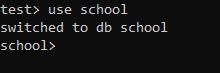


#### CREATE NEW DATABASE

Create new database using the **use** command followed by the desired database name. For example, to switch to a new database named **"school"**, you can type **"use school"**:

**use** school

If the **"school"** database already exists, **"mongosh"** will switch to that database. If the **"school"** database doesn't exist, **"mongosh"** will create it and switch to it.

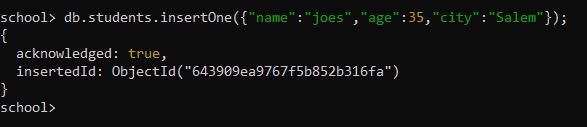


#### HOW TO INSERT NEW RECORDS

You can then start inserting data into the new **"school"** database. For example, you can use the **insertOne** commands to insert documents into collections within the **"school"** database. Here's an example of inserting a document into a **"students"** collection in the **"school"** database:

db.students.insertOne({"name":"Joes","age":30,"city":"Salem"})

This will insert a document with the fields **"name"** , **"age"** and **"city"** into the **"students"** collection in the **"school"** database. If the **"school"** database does not already exist, MongoDB will automatically create it for you.

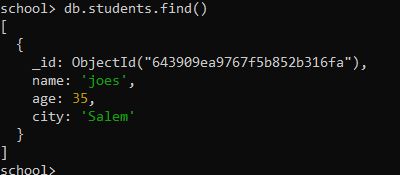


#### HOW TO SELECT RECORDS

Use the **db.collection\_name.find()** method to query data from a specific collection. For example, if you have a collection named **students**, you can use the following command to find all documents in the **students** collection:

db.students.find()

This will return all documents in the **students** collection.

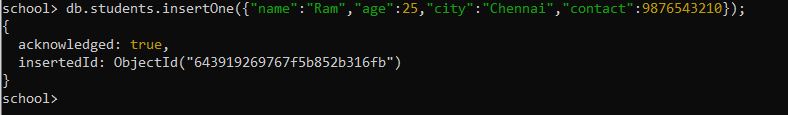


#### HOW TO INSERT ANOTHER NEW RECORDS

You can then start inserting data into the new **"school"** database. For example, you can use the **insertOne** commands to insert documents into collections within the **"school"** database. Here's an example of inserting a document into a **"students"** collection in the **"school"** database:

db.students.insertOne({"name":"Ram","age":25,"city":"Chennai","contact":"9876543210"});

This will insert a document with the fields **"name"** , **"age"**, **"city"** and **"contact"** into the **"students"** collection in the **"school"** database. If the **"school"** database does not already exist, MongoDB will automatically create it for you.

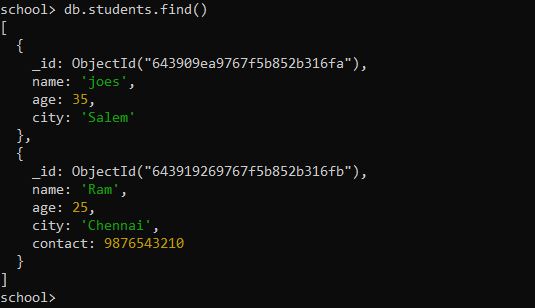


#### HOW TO SELECT RECORDS

Use the **db.collection\_name.find()** method to query data from a specific collection. For example, if you have a collection named **students**, you can use the following command to find all documents in the **students** collection:

db.students.find()

This will return all documents in the **students** collection.

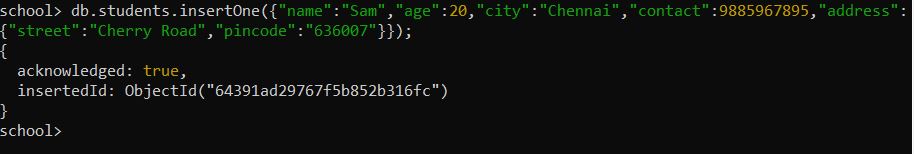


#### HOW TO INSERT ANOTHER NEW RECORDS

You can then start inserting data into the new **"school"** database. For example, you can use the **insertOne** commands to insert documents into collections within the **"school"** database. Here's an example of inserting a document into a **"students"** collection in the **"school"** database:

db.students.insertOne({"name":"Sam","age":20,"city":"Chennai","contact":9885967895,"address":{"street":"Cherry Road","pincode":636007}});

This is the document (object) that you want to insert into the collection. It contains multiple fields such as **name, age, city, contact**, and a nested subdocument **address** with fields **street** and **pincode**.



#### HOW TO SELECT RECORDS

Use the **db.collection\_name.find()** method to query data from a specific collection. For example, if you have a collection named **students**, you can use the following command to find all documents in the **students** collection:

db.students.find()

This will return all documents in the **students** collection.



### **Run MongoDB Queries in Visual Studio Code Terminal**

#### RUN MONGODB QUERIES VSCODE

* Install MongoDB on your system, if it's not already installed. You can download the MongoDB community server from the official website.
* Once MongoDB is installed, open a new terminal window in Visual Studio Code.
* Type **mongo** in the terminal window and press enter. This will start the MongoDB shell.
* To connect to a MongoDB database, type the following command and replace **"database\_name"** with the name of your database: **use database\_name** This will switch to the specified database.
* Now, you can run MongoDB queries in the terminal window. For example, to show all documents in a collection, type: **db.collection\_name.find()** Replace "collection\_name" with the name of your collection.
* You can also run more complex queries using MongoDB's query language. For example, to find documents where the "name" field contains the word "john", type: **db.collection\_name.find({ name: "john" })** This will return all documents where the **"name"** field matches **"john"**.
* Once you are done with the MongoDB shell, type **exit** in the terminal window to close it.

#### CREATE YOUR OWN \_ID IN MONGODB

In MongoDB, if you don't provide an **\_id** field when inserting a document, MongoDB will automatically create one for you. However, if you want to create your own **\_id** field, you can do so by following these steps:

When inserting a new document, include an **\_id** field with a unique value. You can use any value you like, as long as it's unique within the collection.

For example, let's say you want to insert a new document into a collection called **"students"** with an **\_id** of **"1"**. Here's how you would do it:

db.students.insertOne({"\_id":1,"name":"Suresh"})

If you try to insert a document with an **\_id** value that already exists in the collection, MongoDB will return an error. So make sure you choose a unique value for the **\_id** field.

### **How to Insert Array of Datas in Document in MongoDB**

#### INSERT ARRAY OF DATAS IN DOCUMENT

To insert an array of data into a document in MongoDB, you can use the **insertOne()** method with an object that contains an array field. The array field can then contain any number of elements, including strings, numbers, objects, or other arrays.

Here's an example query that inserts a document with an array of hobbies into a collection called **"students"**:

db.students.insertOne({"\_id":2,"name":"Sara","hobbies":["Dancing","Drawing","Singing"]});

In this example, we're inserting a document with an **\_id** of **2**, a **name** of **"Sara"**, and an array of **hobbies** that includes **"Dancing", "Drawing", and "Singing"**.

If you want to insert multiple documents with arrays, you can use the **insertMany()** method instead of **insertOne()**, and pass an array of objects to the method.

You can now insert an array of data into a document in MongoDB using the **insertOne()** method.

#### VISUAL STUDIO CODE EXTENSION FOR MONGODB

To install a Visual Studio Code extension for MongoDB, you can follow these steps:

* Open Visual Studio Code and click on the Extensions icon in the left-hand sidebar (it looks like four squares).
* In the Extensions search bar at the top of the screen, type in the name of the MongoDB extension you want to install (e.g. "MongoDB for VS Code").
* From the search results, select the extension you want to install and click on the **"Install"** button.
* Once the installation is complete, you may need to restart Visual Studio Code for the extension to be fully activated.
* Once the extension is installed and activated, you should be able to access its features and functionality from within Visual Studio Code.

#### CREATE A NEW PLAYGROUND FOR MONGODB

To create a new playground for MongoDB in Visual Studio Code using the MongoDB for VS Code extension, you can follow these steps:

* Open Visual Studio Code and navigate to the Explorer sidebar. Right-click in the Explorer pane and select "New File" to create a new file.
* Save the file with a .mongo extension (e.g. myplayground.mongo) to indicate that it is a MongoDB playground file.
* Open the file and type the following code to connect to your MongoDB database and execute queries:

### **How to Insert Multiple Documents using insertMany()**

#### HOW TO INSERT MULTIPLE RECORDS

This **insertMany()** is a MongoDB method that allows you to insert multiple documents into a collection in one operation. Here's an example of how to use **insertMany()** to insert multiple documents into the **students** collection:

db.students.insertMany([

{"\_id":3,"name":"Ramesh"},

{"\_id":4,"name":"Tiya","age":15},

{"name":"Diya","age":5}

]);

In this example, we are inserting three documents into the **students** collection:

* The first document has an explicitly defined **\_id** field with a value of **3** and a **name** field with a value of **"Ramesh"**.
* The second document also has an explicitly defined **\_id** field with a value of **4**, a **name** field with a value of **"Tiya"**, and an **age** field with a value of **15**.
* The third document has no **\_id** field defined, so MongoDB will automatically generate an **\_id** field for it, and it has a **name** field with a value of **"Diya"** and an **age** field with a value of **5**.

Note that if any of the documents in the array already have an **\_id** field that matches an existing document in the collection, MongoDB will throw a duplicate key error and the entire **insertMany()** operation will fail.

#### HOW TO SELECT RECORDS

db.students.find({},{"\_id":0,"name":1,"age":1});

The MongoDB query **db.students.find({},{"\_id":0,"name":1,"age":1})** finds all documents in the **students** collection and returns only the **name** and **age** fields for each document, while excluding the **\_id** field. Here's how this query works:

* **db.students**: specifies the name of the collection to search in (in this case, students).
* **.find()**: specifies that we want to find documents in the collection that match a certain criteria.
* **{},**: provides an empty filter expression, which matches all documents in the collection.
* **{"\_id":0,"name":1,"age":1}**: specifies which fields we want to include or exclude from the result set. In this case, we're excluding the \_id field (since "0" means exclude) and including the**name and age** fields (since "1" means include).

So, when you run this query in the MongoDB shell, it will return a cursor that contains all documents in the **students** collection, but with only the **name and age** fields included, and the **\_id** field excluded.

#### Source Code

use("school");

*//show dbs;*

*//use school;*

db.getMongo().getDBs();

db.getCollectionNames();

db.students.insertOne({"name":"joes","age":35,"city":"Salem"});

db.students.find();

db.students.insertOne({"name":"Ram","age":25,"city":"Chennai","contact":"9876543210"});

db.students.find();

db.students.insertOne({"name":"Sam","age":20,"city":"Chennai","contact":"9885967895","address":{"street":"cherry Road","pincode":"636007"}});

db.students.find();

db.students.find({ name: "joes" });

db.students.insertOne({"\_id": 2,"name": "Sara","hobbies": ["Dancing","Drawing", "Singing" ]});

db.students.insertMany([

{"\_id":3,"name":"Ramesh"},

{"\_id":4,"name":"Tiya","age":15},

{"name":"Diya","age":5}

]);

db.students.find({},{"\_id":0,"name":1,"age":1});

This is a sequence of MongoDB shell commands that perform various operations on the **students** collection within the **school** database. Here's an explanation of each command:

* use("school"): selects the school database as the current database.
* db.getMongo().getDBs() and db.getCollectionNames(): These two commands retrieve the list of all databases and collections, respectively, that are accessible by the current connection.
* db.students.insertOne({"name":"joes","age":35,"city":"Salem"}): Inserts a new document into the students collection with the specified fields.
* db.students.find(): Returns all documents in the students collection.
* db.students.insertOne({"name":"Ram","age":25,"city":"Chennai","contact":"9876543210"}): Inserts a new document with an additional field, contact, into the students collection.
* db.students.insertOne({"name":"Sam","age":20,"city":"Chennai","contact":"9885967895","address":{"street":"cherry Road","pincode":"636007"}}): Inserts a new document with an additional nested field, address, into the students collection.
* db.students.find({ name: "joes" }): Returns all documents in the students collection that have a name field equal to "joes".
* db.students.insertOne({"\_id": 2,"name": "Sara","hobbies": ["Dancing","Drawing", "Singing" ]}): Inserts a new document into the students collection with a manually specified \_id field and an additional field, hobbies, which is an array.
* db.students.insertMany([...]): Inserts multiple documents into the students collection using the insertMany() method.
* db.students.find({},{"\_id":0,"name":1,"age":1}): Returns all documents in the students collection, but only with the name and age fields included, and the \_id field excluded.

### **Managing MongoDB Databases: Create, Show, and Delete Operations**

#### Create Database

In MongoDB, you don't explicitly create a database like you would in a relational database system. MongoDB is a NoSQL database, and databases and collections (equivalent to tables in relational databases) are created automatically when you insert data into them.

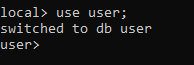
#### Switch to a Database

To work with a specific database, you can use the use command. For example, to switch to a database named "user," you can type.

**Here are the basic steps to create a database in MongoDB:**

**use** user

#### Output



#### Show all Database

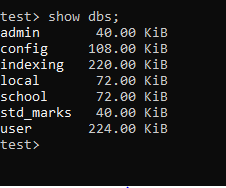
In MongoDB, you can list all the databases by using the**show dbs** command

In the MongoDB , simply type the following command and press Enter:

**Verify Database Creation**: You can check if the database and collection have been created by running:

**show** dbs

#### Output



Remember that in MongoDB, a database is only created when you insert data into it or perform some operations that require the database to exist. You don't need to explicitly create a database schema as you would in a traditional relational database. Just start inserting data into the desired database, and MongoDB will create it if it doesn't exist.

#### Create Collection inside a Database

**Switch to the Database:** Use the use command to switch to the database where you want to create the collection. Replace "user" with the name of your target database:

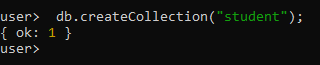
**Syntax**

db.createCollection("collectionName")

**Example**

db.createCollection("student")

#### Output



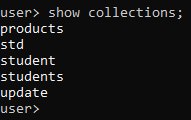
If the collection is successfully created, MongoDB will acknowledge it with a**{ "ok" : 1 }**response.

#### Show Collection Database

**Verify Collection Creation**: You can verify that the collection has been created by listing all the collections in the current database using the show collections command

**show** collections

#### Output



You have now created a collection named **"collections"** within the **"user"** database. You can start inserting documents into this collection and performing various operations on it.

Keep in mind that this command only lists collections in the currently selected database. If you want to see collections in a different database, you'll need to use the **use** command to switch to that database and then run **show collections** again.

#### Delete Database

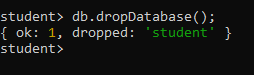
To delete a database in MongoDB, you can use the **db.dropDatabase()** method in the MongoDB.

**Warning:** Deleting a database in MongoDB is a permanent operation, and all data in the database will be lost. Be cautious when using this command, and make sure you have a backup if necessary.

Once you are in the correct database, run the following command to delete it:

db.dropDatabase()

#### Output



MongoDB will prompt you with a warning message to confirm the deletion. Type **1** or **true** to confirm and delete the database:

{ok:1, dropped:'student'}

Please exercise caution when using the dropDatabase command, as it permanently removes all data in the specified database, and there is no way to recover it. Make sure you have a backup of your data if needed before proceeding with the deletion.

### **Inserting Documents into MongoDB Collections: A Step-by-Step Guide**

#### Insert a Single Record into the Collections with Id

In MongoDB, when you insert a document into a collection without explicitly specifying an **\_id** field, MongoDB will automatically generate a unique \_id for that document. However, if you want to insert a single record into a collection with a specific **\_id** value, you can do so by including the **\_id** field in your document.

**Syntax**

db.collectionName.insertOne()

Here's how to insert a single record with a specific **\_id** into a MongoDB collection:

db.students.insertOne({

"\_id":1,

"Name":"Sam",

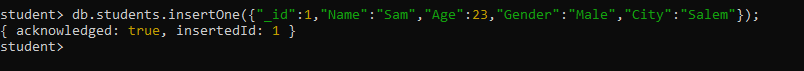
"Age":23,

"Gender":"Male",

"City":"Salem"

})

#### Output



**insertOne({ "\_id": 1, "Name": "Sam", "Age": 23, "Gender": "Male", "City": "Salem" }):** This is the **insertOne()** method call, which is used to insert a single document into the collection. It includes an object (document) as an argument, and this document represents the data you want to insert

* "\_id": 1: This field specifies the unique identifier (ID) for the document. In this case, you've explicitly set it to 1. MongoDB will use this value as the document's primary key. If you insert another document with the same **\_id**, it will overwrite the existing document with that ID.
* "Name": "Sam": This field contains the name of the student, which is "Sam."
* "Age": 23: This field represents the age of the student, which is 23.
* "Gender": "Male": This field indicates the gender of the student, which is "Male."
* "City": "Salem": This field specifies the city where the student resides, which is "Salem."

Once you run this MongoDB shell command, it will insert the provided document into the "students" collection with the specified fields and values. The document will have an **\_id** of 1.

In summary, this code inserts a single student record with specific information into the "students" collection, and it assigns an explicit **\_id** value of 1 to the document. The **\_id** field is important because it ensures the uniqueness of documents within the collection and serves as the primary key for retrieval and update operations.

#### Insert a Single Record into the Collections without Id(It is Self-Generated)

In MongoDB, if you want to insert a single record into a collection without specifying a value for the **\_id** field, MongoDB will automatically generate a unique **\_id** for that document. You can achieve this by simply omitting the **\_id** field when inserting the document.

Assuming you have a collection named "mycollection" and you want to insert a document without specifying an **\_id**:

db.students.insertOne({

"Name":"Sam",

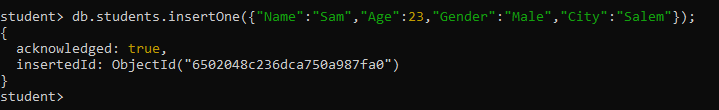
"Age":23,

"Gender":"Male",

"City":"Salem"

})

#### Output



#### Insert a Multiple Record into the Collections without Id(It is Self-Generated)

To insert multiple records into a MongoDB collection without specifying an explicit **\_id**(thus allowing MongoDB to generate unique **\_id**values automatically), you can use the **insertMany()** method.

db.collectionName.insertMany()

Assuming you have a collection named "students," and you want to insert multiple documents without specifying **\_id**values:

db.students.insertMany([

{

"Name":"Sam",

"Age":23,

"Gender":"Male",

"City":"Salem"

},

{

"Name":"Ram",

"Age":33,

"Gender":"Male",

"City":"Coimbatore"

},

{

"Name":"Sara",

"Age":21,

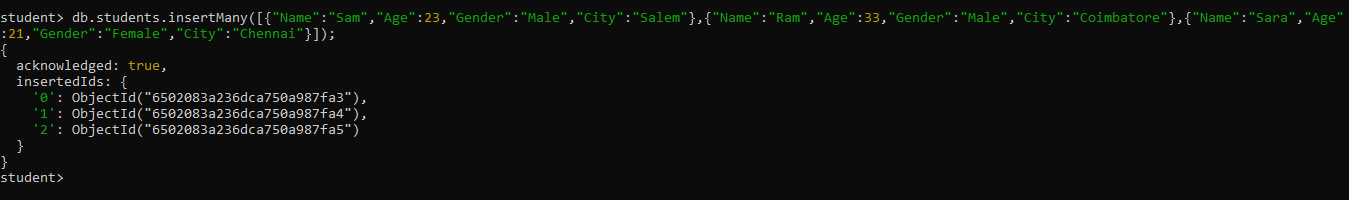
"Gender":"Female",

"City":"Chennai"

}

])

#### Output



#### Insert a Multiple Record into the Collections with Id

To insert multiple records into a MongoDB collection with explicitly specified **\_id** values, you can use the **insertMany()** method and include the **\_id** field in each document.

db.collectionName.insertMany()

Assuming you have a collection named "**student**," and you want to insert multiple documents with specific **\_id** values

db.students.insertMany([

{

"\_id":1,

"Name":"Sam",

"Age":23,

"Gender":"Male",

"City":"Salem"

},

{

"\_id":2,

"Name":"Ram",

"Age":33,

"Gender":"Male",

"City":"Coimbatore"

},

{

"\_id":3,

"Name":"Sara",

"Age":21,

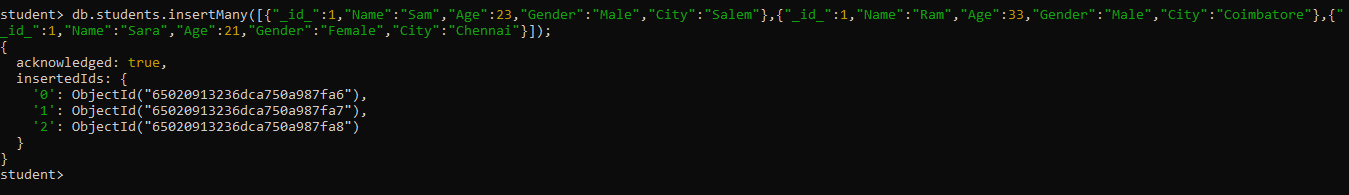
"Gender":"Female",

"City":"Chennai"

}

])

#### Output



we've provided an array of documents to the **insertMany()** method, and each document includes an **\_id** field with a specific value. MongoDB will use these specified **\_id** values for the inserted documents.

Please replace **"student"** with the actual name of your collection, and adjust the **\_id** values and other document data as needed for your specific use case. Keep in mind that each **\_id** value must be unique within the collection to avoid conflicts.

### **Performing Select or Find Queries in MongoDB: A Comprehensive Guide**

#### Select or Find query

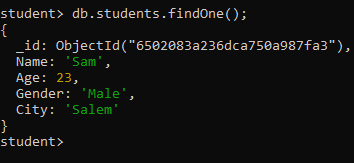
To retrieve the first record (document) from a collection in MongoDB, you can use the **findOne()** method. Here's how you can do it.

db.collectionName.findOne()

Assuming you have a collection named**"student,"** you can retrieve the first document from it like this

db.students.findOne()

#### Output



Keep in mind that the order of documents in MongoDB is not guaranteed unless you have explicitly sorted them in a specific order. If you need a specific document to be first, you should apply sorting or filtering criteria to your query to ensure the desired result.

#### Show or Select a particular Collection

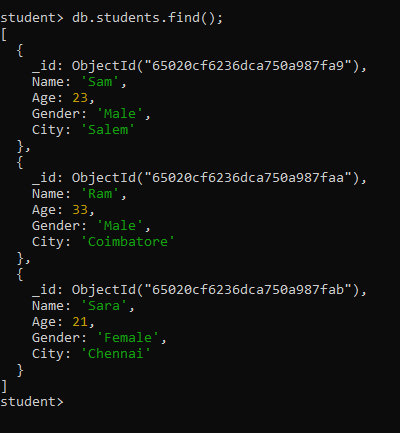
In MongoDB, to select or show the contents of a particular collection, you can use the**db.collectionName.find()**command in the MongoDB. Here's how you can do it:

db.collectionName.find()

Once you are in the correct database, you can use the db.collectionName.find() command to select or show the contents of the desired collection. Replace "collectionName" with the name of the collection you want to work with

db.students.find()

#### Output



#### Count of Field or Record in Particular Collection

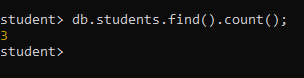
To count the number of documents (records) in a particular collection in MongoDB, you can use the countDocuments() method. Here's how you can do it

db.collectionName.find().count()

Assuming you have a collection named "students," you can count the documents in it like this:

db.students.find().count()

#### Output



#### To find a Particular Field in Collection

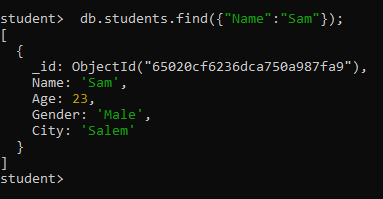
To find a particular field in a collection in MongoDB, you can use the find() method and specify the field you want to retrieve in the projection. Here's how you can do it

db.mycollection.find({}, { fieldName: value })

If you want to filter or query specific documents within the collection, you can provide query criteria as an argument to the find() method. For example, to find documents where the**"Name"** field is **"Sam,"** you can do

db.students.find({"Name":"Sam"})

#### Output



#### Using Count for Particular Field

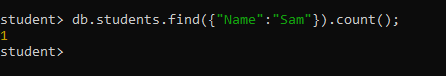
If you want to count the occurrences of a particular field for documents that match specific criteria in MongoDB, you can use the **count()** method along with a query. Here's how you can do it

db.collectionName.find({"feildName"}).count()

Assuming you have a collection named **"students,"** and you want to count the occurrences of the "fieldName" field for documents where a certain condition is met:

db.students.find({"Name":"Sam"}).count()

#### Output



#### Using Limit

In MongoDB, you can use the **limit()** method to limit the number of documents returned by a query. This is especially useful when you want to retrieve only a specific number of documents from a collection. Here's how to use **limit()**

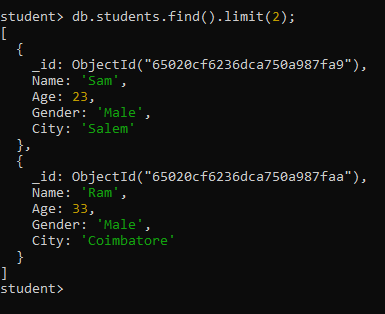
db.collectionName.find(query, projection).**limit**(limitValue)

* **collectionName:** Replace this with the name of your collection.
* **query:** Optional. You can specify query conditions here to filter the documents you want to retrieve. If you don't specify a query, it will match all documents in the collection.
* **projection:** Optional. You can specify which fields you want to include or exclude from the result documents. If you don't specify a projection, it will include all fields.
* **limitValue:** The number of documents to limit the query results to.

**Select first 2 or more records (Ascending Order)**

db.students.find().**limit**(2)

#### Output



**Select first 2 or more records (Descending Order)**

db.students.find().**limit**(-2)

Keep in mind that the **limit()** method is typically used for pagination or to limit the number of results when you don't want to retrieve the entire collection. It should be used in combination with other query operators to achieve more complex filtering and sorting if needed.

#### Sort

In MongoDB, the **sort()** method is used to sort the documents in a collection based on one or more fields. You can use **sort()** to arrange the documents in either ascending (ascending order) or descending (descending order) fashion. Here's how to use **sort().**

db.collectionName.find(query).sort(sortSpecification)

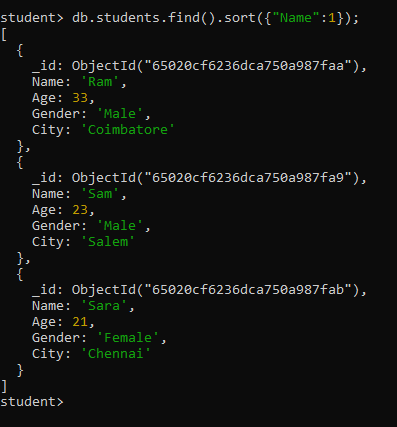
* **collectionName:** Replace this with the name of your collection.
* **query:** Optional. You can specify query conditions here to filter the documents you want to retrieve. If you don't specify a query, it will match all documents in the collection.
* **sortSpecification:** An object that specifies the fields by which you want to sort and the sorting order. It should be in the format**{ fieldName: sortOrder }**, where fieldName is the name of the field to sort by, and sortOrder can be **1** for ascending or **-1** for descending.

**Ascending Order**

Sort documents in the "students" collection in ascending order based on the "Name" field.

db.students.find().sort({"Name":1})

#### Output

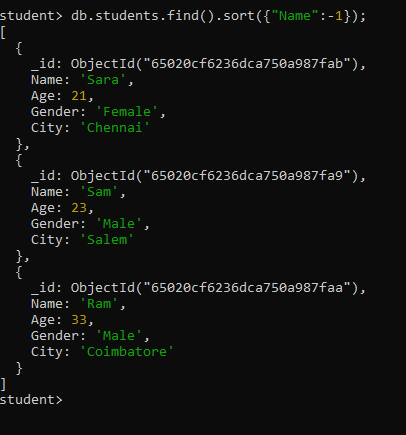


**Descending Order**

Sort documents based on multiple fields. For example, sort by "name" in descending order.

db.students.find().**sort**({"Name":-1})

#### Output



#### Show and Hide a Particular Field in Collections

In MongoDB, you can control which fields are shown or hidden (excluded) in query results using the projection feature. You can use the **find()** method with projection to specify which fields you want to include or exclude. Here's how you can show and hide a particular field in collections

db.collectionName.find({"condition"},{"feildName":value})

**Show a Particular Field**

To show a particular field, include the field you want to display in the projection.For example, if you have a collection named "students" and you want to show the **"name"** and **"gender"** field.

db.students.find({"Gender":"Male"},{"Gender":1,"Name":1,"\_id":0})

In this example, **{ gender:1, name: 1, \_id: 0 }** is the projection. It includes the "name" and "gender" field (1 indicates inclusion), and it excludes the "\_id" field (0 indicates exclusion). The **"\_id"** field is automatically included unless explicitly excluded.

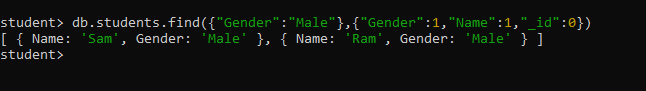
**Hide a Particular Field**

To hide a particular field, simply exclude it from the projection. For example, if you want to hide the "id" field in the query results

db.students.find({"Gender":"Male"},{"Gender":1,"Name":1,"\_id":0})

In this example, { id: 0 } is the projection, and it excludes the "id" field from the query results.

#### Output



You can customize the projection to include or exclude multiple fields based on your requirements.

#### Distinct

In MongoDB, the **distinct()** method is used to retrieve distinct values (unique values) from a specified field in a collection. It is often used when you want to find unique values within a specific field, such as finding all unique categories in a collection.

Here's the basic syntax for using distinct():

db.collectionName.**distinct**(fieldName, query)

* **collectionName:** Replace this with the name of your collection.
* **fieldName:**Specify the field from which you want to retrieve distinct values.
* **query:** Optional. You can provide a query to filter the documents before finding distinct values. This is useful when you want to find distinct values only among a subset of documents.

**Data**

[

{

"Name":"Mano Bala",

"Age":25,

"Blood\_Group":"B+",

"Gender":"Male",

"Pressure\_Level":"140/100mmHg"

},

{

"Name":"Renuka",

"Age":27,

"Blood\_Group":"O-",

"Gender":"Female",

"Pressure\_Level":"110/90mmHg"

},

{

"Name":"Menaga",

"Age":26,

"Blood\_Group":"AB-",

"Gender":"Female",

"Pressure\_Level":"130/100mmHg"

},

{

"Name":"Jaanu",

"Age":23,

"Blood\_Group":"A+",

"Gender":"Female",

"Pressure\_Level":"120/80mmHg"

}

]

db.user.**distinct**("Blood\_Group",{"Gender":"Female"})

The code you provided is a MongoDB query that uses the **distinct()** method to find distinct values in the "Blood\_Group" field for documents where the "Gender" field is "Female" in the "user" collection. Let's break down the code step by step

* **db.user:** This part specifies the collection you are querying. In this case, it's the "user" collection
* **distinct("Blood\_Group", {"Gender": "Female"})** This is the **distinct()** method call, and it has two arguments:
  1. **"Blood\_Group":**This argument specifies the field for which you want to find distinct values. In this case, you want to find distinct values in the "Blood\_Group" field.
  2. **{"Gender": "Female"}:** This argument is a query filter that restricts the documents considered for finding distinct values. It means you only want to find distinct "Blood\_Group" values for documents where the "Gender" field is equal to "Female."

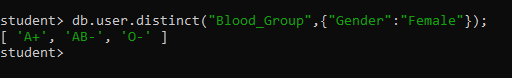
The result of this query will be an array containing the unique **"Blood\_Group"** values from the documents that meet the specified filter criteria.

**Field Only**

The code **db.user.distinct("Blood\_Group")** is a MongoDB query that uses the **distinct()** method to retrieve distinct values from the "Blood\_Group" field in the "user" collection.

db.user.**distinct**("Blood\_Group")

#### Output



* **db.user:** This part specifies the collection you are querying, which is the "user" collection in this case.
* **.distinct("Blood\_Group"):** This is the distinct() method call, and it has one argument:
  + **"Blood\_Group":** This argument specifies the field for which you want to find distinct values, in this case, "Blood\_Group."

When you run this query, MongoDB will return an array containing all the unique values found in the **"Blood\_Group"**field across all documents in the "user" collection.

This can be useful when you want to obtain a list of unique values from a specific field within a collection, which can be helpful for analysis or generating summary information.

#### Skip

In MongoDB, you can use the **skip()** method in conjunction with the **find()** method to skip a specified number of documents and retrieve documents starting from a particular position in the result set. This is often used for pagination or to skip a certain number of documents before fetching the next set of results. Here's how to use it

The basic syntax for using **skip()** is as follows:

db.collectionName.find(query).skip(numberOfDocumentsToSkip)

* **collectionName:**Replace this with the name of your collection.
* **query:**Optional. You can specify query conditions here to filter the documents you want to retrieve. If you don't specify a query, it will match all documents in the collection.
* **numberOfDocumentsToSkip:**The number of documents to skip in the result set before starting to fetch documents.

**Data**

[

{

"name": "Sundar",

"hobbies": [

{

"title": "Sports",

"frequency": 3

},

{

"title": "Cooking",

"frequency": 6

}

],

"phone": 131782734

},

{

"name": "Kumar",

"hobbies": [

{

"title": "Cooking",

"frequency": 5

},

{

"title": "Cars",

"frequency": 2

}

],

"phone": "012177972",

"age": 30

},

{

"name": "Sra",

"hobbies": [

{

"title": "Sports",

"frequency": 2

},

{

"title": "Yoga",

"frequency": 3

}

],

"phone": "8958785858",

"age": null

},

{

"name": "Raja",

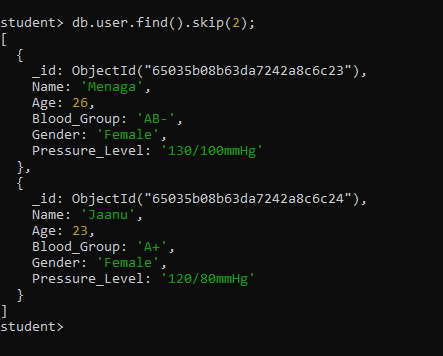
"hobbies": ["Sports", "Cooking", "Hiking"]

}

]

db.user.find().skip(2)

#### Output



The code **db.user.find().skip(2)** is a MongoDB query that retrieves documents from the "user" collection but skips the first two documents in the result set.

* **db.user:** This part specifies the collection you are querying, which is the "user" collection in this case.
* **.find():** This is the **find()** method used to query the collection. In this example, it retrieves all documents in the collection without any specific filtering criteria (i.e., it matches all documents).
* **.skip(2):** This is the **skip()** method, which specifies that you want to skip the first 2 documents in the result set.

When you run this query, MongoDB will retrieve all documents in the "user" collection but exclude the first two documents from the result. The result will include all documents starting from the third document onwards.

This can be useful for scenarios where you want to implement pagination or retrieve a subset of documents from a larger collection.

#### Check Key Exists or not

In MongoDB, you can check if a specific field exists in a document using various query operators and methods. Here are some common approaches to check if a key (field) exists in a document

db.collectionName.find({"feildName":{$exists:true or false}})

**Using the $exists Operator**

You can use the **$exists**operator within a query to check if a field exists in a document. It returns documents where the specified field either exists **(true)** or does not exist **(false)**. For example:**if true**

db.user.find({"Name":{$exists:true}})

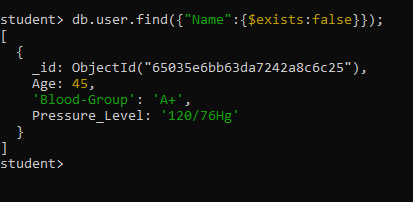
#### Output



**if false**

db.user.find({"Name":{$exists:false}})

#### Output



### **Finding Documents by Object Data in MongoDB**

In MongoDB, you can store nested objects (also known as embedded documents) within a document. To represent titles and keywords as nested objects within a document, you can structure your data like this

#### Example

[

{

"Name":"Siva",

"address":{

"district":"Salem",

"Pincode":12345,

},

"City":"chennai"

},

{

"Name":"Mani",

"address":{

"district":"Trichy",

"Pincode":12345,

},

"City":"Madhurai"

}

]

**Find**

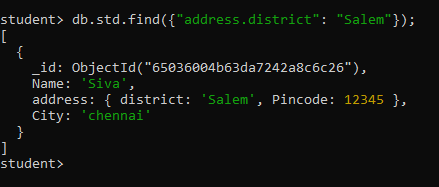
db.std.find({"address.district": "Salem"})

The code snippet you provided is a query written in the MongoDB query language to find documents in a collection named "std" where the value of the "district" field within the "address" subdocument is equal to "Salem."

* **db.std.find:**This is the MongoDB "find" operation. It is used to search for documents in the "std" collection.
* **{"address.district": "Salem"}:** This is the query criteria enclosed in curly braces. It specifies the condition that documents must meet to be considered a match. In this case, it's looking for documents where the value of the "district" field within the "address" subdocument is equal to "Salem."

So, when you run this code against your MongoDB database, it will return all documents in the "std" collection where the "district" field in the "address" subdocument is "Salem."

#### Output



**Count**

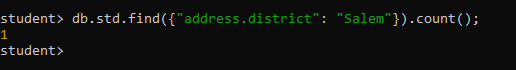
db.std.find({"address.district": "Salem"}).count()

The code you provided is an extension of the previous code snippet. It's a MongoDB query that finds documents in the "std" collection where the value of the "district" field within the "address" subdocument is equal to "Salem," and then it counts the number of matching documents. Here's an explanation:

* **db.std.find({"address.district": "Salem"}):**This part of the code performs a find operation in the "std" collection. It searches for documents that meet the specified criteria, where the value of the "district" field within the "address" subdocument is equal to "Salem."
* **.count():** After the find operation, the **count()**method is called on the result of the find operation. This method counts the number of documents that match the specified query criteria.

So, when you run this code against your MongoDB database, it will return the count (the number) of documents in the "std" collection where the "district" field in the "address" subdocument is "Salem."

#### Output



**Limit**

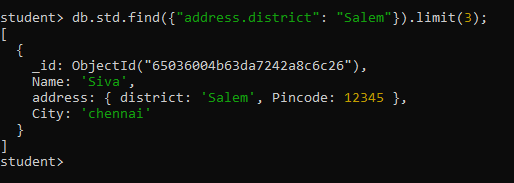
The code you provided is a MongoDB query that finds documents in the "std" collection where the value of the "district" field within the "address" subdocument is equal to "Salem" and then limits the result to return only the first 3 matching documents. Here's an explanation

db.std.find({"address.district": "Salem"}).**limit**(3)

* **db.std.find({"address.district": "Salem"}):**This part of the code performs a find operation in the "std" collection. It searches for documents that meet the specified criteria, where the value of the "district" field within the "address" subdocument is equal to "Salem."
* **.limit(3):** After the find operation, the **limit()** method is called on the result of the **find** operation. This method limits the number of documents returned in the result set to a specified number, in this case, 3.

So, when you run this code against your MongoDB database, it will return the first 3 documents in the "std" collection where the "district" field in the "address" subdocument is "Salem." If there are more than 3 matching documents, only the first 3 will be included in the result.

#### Output



**Limit and Count**

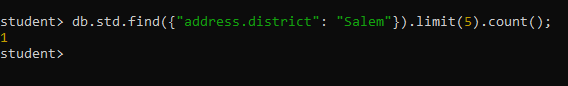
db.std.find({"address.district": "Salem"}).**limit**(5).count()

The code you provided is a MongoDB query that finds documents in the "std" collection where the value of the "district" field within the "address" subdocument is equal to "Salem," limits the result to return only the first 5 matching documents, and then counts the number of documents in this limited result. Here's an explanation:

* **db.std.find({"address.district": "Salem"}):** This part of the code performs a find operation in the "std" collection. It searches for documents that meet the specified criteria, where the value of the "district" field within the "address" subdocument is equal to "Salem."
* **.limit(5):** After the **find** operation, the **limit()** method is called on the result of the find operation. This method limits the number of documents returned in the result set to a specified number, in this case, 5.
* **.count():**After limiting the result to 5 documents, the **count()** method is called. It counts the number of documents in the limited result set.

So, when you run this code against your MongoDB database, it will return the count of the first 5 documents in the "std" collection where the "district" field in the "address" subdocument is "Salem." If there are more than 5 matching documents, only the first 5 will be included in the count.

#### Output



**Limit and Skip**

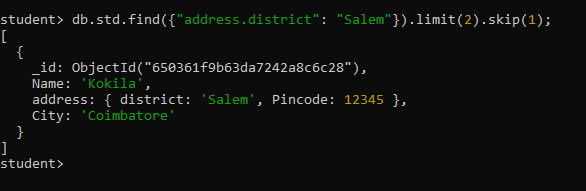
The code you provided has a **.skip(-1)**operation, which is not typically valid in MongoDB. The **.skip()**method is used to skip a specified number of documents from the beginning of the result set. A negative value for **.skip()** is not allowed because it would not make logical sense to skip a negative number of documents.

db.std.find({"address.district": "Salem"}).**limit**(2).skip(-1)

* **db.std.find({"address.district": "Salem"}):** This part of the code performs a find operation in the "std" collection. It searches for documents that meet the specified criteria, where the value of the "district" field within the "address" subdocument is equal to "Salem."
* **.limit(2):** After the find operation, the .limit() method is called, which limits the result to return only the first 2 matching documents.
* **.skip(-1):**The .skip() method is called with a negative value of -1, which is not a valid operation. Skipping a negative number of documents doesn't make sense in the context of a database query.

If you intended to skip a specific number of documents, you should use a positive integer value with **.skip()**. For example, **.skip(3)** would skip the first 3 documents in the result set. If you have a specific use case in mind, please provide more details, and I can assist you further.

#### Output



### **Slicing Arrays in MongoDB Documents**

The "slice" operation in MongoDB allows you to extract a specified number of elements from an array field within a document. This operation is particularly useful when you want to retrieve a subset of elements from an array without fetching the entire array. It can be handy for pagination or when you need to work with a specific range of elements within an array, such as the most recent items or a specific page of results.

**Example**

[

{

"Name":"Vinoth",

"Age":23,

"Gender":"Male",

"Skill":["c","c++","java","python"],

"University":"Anna University "

},

{

"Name":"Deppak",

"Age":25,

"Gender":"Male",

"Skill":["javaScript","MYSQL","bootstrap-5","mongo DB"],

"University":"Periyar University "

}, {

"Name":"Sara",

"Age":28,

"Gender":"Female",

"Skill":["SQL","express js","react js","node js"],

"University":"Annamalai University "

}

]

**SYNTAX**

db.collectionName.find({"condition"}{"feildName":{$slice:**index** value}})

The code you provided is a MongoDB query that searches for documents in a collection named "data" where the "Name" field is equal to **"Sara."** It then projects the "Skill" field using the **$slice**operator to retrieve only the first 2 elements from the **"Skill"** array within each matching document. Here's a breakdown of the code

db.**data**.find({"Name":"Sara"},{"Skill":{$slice:2}})

* **db.data.find({"Name":"Sara"}):**This part of the code performs a find operation in the "data" collection. It searches for documents that meet the specified criteria, where the "Name" field is equal to "Sara."
* **{"Skill": {$slice: 2}}:**After the find operation, the projection part of the query specifies which fields to include in the result. In this case, it's specifying that only the "Skill" field should be included. Additionally, the $slice operator is applied to the "Skill" field, indicating that only the first 2 elements of the "Skill" array should be included in the result.

#### Output

So, when you run this code against your MongoDB database, it will return all documents where "Name" is "Sara," and for each of those documents, it will include only the first 2 elements of the "Skill" array in the result. The result will be a list of documents with "Name" equal to "Sara," and each document will have a "Skill" array containing at most 2 elements.

**Descending Order in Slicing**

Filter or remove First two index value

db.collectionName.find({"condition"}{"feildName":{$**slice**:-index value}})

The below code you provided is a MongoDB query that searches for documents in a collection named "data" where the "Name" field is equal to "Vinoth." It then projects the "Skill" field using the $slice operator with a negative value of -2.

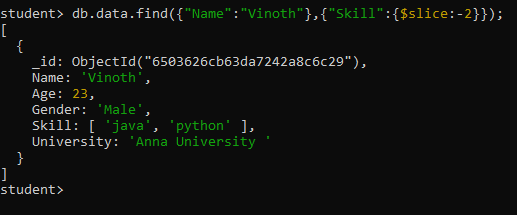
db.data.find({"Name":"Vinoth"},{"Skill":{$**slice**:-2}})

* **db.data.find({"Name":"Vinoth"}):** This part of the code performs a find operation in the "data" collection. It searches for documents that meet the specified criteria, where the "Name" field is equal to "Vinoth."
* **{"Skill": {$slice: -2}}:**After the find operation, the projection part of the query specifies which fields to include in the result. In this case, it's specifying that only the "Skill" field should be included. The $slice operator is applied to the "Skill" field with a negative value of -2.

When you use a negative value with $slice, it indicates that you want to include the last N elements of the array, where N is the absolute value of the negative number. In this case, it includes the last 2 elements of the "Skill" array.

So, when you run this code against your MongoDB database, it will return all documents where "Name" is "Vinoth," and for each of those documents, it will include only the last 2 elements of the "Skill" array in the result.

#### Output



**Parameter given in slice**

In MongoDB, the $slice operator is used to project a specified number of elements from an array field within a document. The $slice operator takes one or two arguments:

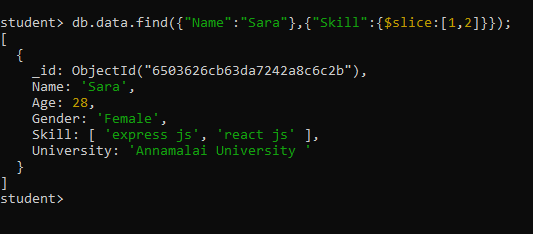
db.collectionName.find({"condition"}{"feildName":{$**slice**:[starting index value,last index value]}})

Number of Elements: If you provide a single argument, it represents the number of elements you want to project from the beginning of the array.

Skip and Limit: If you provide two arguments, they represent the number of elements to skip and the number of elements to project, respectively. The first argument specifies the number of elements to skip from the beginning of the array, and the second argument specifies the number of elements to project.

db.data.find({"Name":"Sara"},{"Skill":{$**slice**:[1,2]}})

#### Output



**Using Dollar Sign**

The dollar sign **($)** is used to indicate that MongoDB should project the specific matching element from the array.

**Example**

db.std.find(

{"languages" : "Tamil"},

{"languages.$" :1}

)

The MongoDB query you provided searches for documents in the "std" collection where the "languages" array contains the element "Tamil." It then projects the "languages" array element that matched the query using **"languages.$": 1**.

* **db.std.find(...):** This part of the code performs a find operation in the "std" collection.
* **{"languages": "Tamil"}:** This is the query criteria. It specifies that the documents to be retrieved must have an array field named "languages" that contains the element "Tamil."
* **{"languages.$": 1}:** This is the projection part of the query. It specifies that only the "languages" array element that matched the query (in this case, "Tamil") should be included in the result. The dollar sign **($)** is used to indicate that MongoDB should project the specific matching element from the array.

So, when you run this code against your MongoDB database, it will return all documents in the "std" collection where the "languages" array contains the element "Tamil," and for each of those documents, it will include only the "languages" array element that matched the query in the result.

**Using ALL in array**

**Syntax**

db.collectionName.find({"feildName":{$all:["index value" , "index value"]}})

[

{

"Name":"Sachin",

"Language":["Tamil","English"],

"Age":25,

"Gender":"Male"

},

{

"Name":"Virat",

"Language":["Hindi","English"],

"Age":34,

"Gender":"Male"

},

{

"Name":"Gill",

"Language":["Hindi","Tamil"],

"Age":29,

"Gender":"Male"

},

{

"Name":"Guna",

"Language":["Tamil","English"],

"Age":30,

"Gender":"Male"

},

{

"Name":"Mandhana",

"Language":["Tamil","English"],

"Age":25,

"Gender":"Female"

}

]

db.std.find(

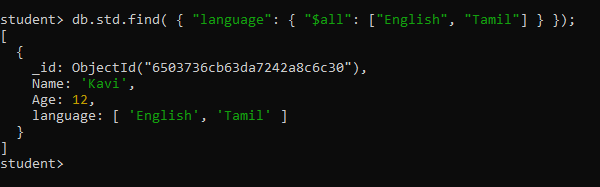
{"languages":{

"$all" :[ "English", "Tamil"]

}}

)

#### Output



The MongoDB query you provided searches for documents in the "std" collection where the "languages" array contains both "English" and "Tamil."

* **db.std.find(...):** This part of the code performs a find operation in the "std" collection.
* **{"languages": { "$all": ["English", "Tamil"] }}:** This is the query criteria. It specifies that the documents to be retrieved must have an array field named "languages" that contains all the elements listed in the $all operator. In this case, it's looking for documents where the "languages" array contains both "English" and "Tamil."

The $all operator is used to match documents where the specified array field contains all the elements provided in the array (["English", "Tamil"] in this case). It ensures that the document must contain both "English" and "Tamil" within its "languages" array for it to be considered a match.

So, when you run this code against your MongoDB database, it will return all documents in the "std" collection where the "languages" array contains both "English" and "Tamil."

**Finding an Array by an Element**

To find documents in MongoDB where an array contains a specific element

The MongoDB query syntax you provided uses the $and operator to combine multiple conditions in a query. Here's an explanation of the**syntax**:

db.collectionName.find({

$and: [

{"fieldName1": "condition1"},

{"fieldName2": "condition2"}

]

})

* **db.collectionName.find(...):** This part of the code performs a find operation in the specified collection (collectionName).
* **$and:** The $and operator is used to specify that all the conditions inside it must be true for a document to be considered a match. In this case, it's used to combine two conditions.
* **{"fieldName1": "condition1"}:** This is the first condition. It specifies that you want to find documents where the field named "fieldName1" matches the specified condition ("condition1").
* **{"fieldName2": "condition2"}:**This is the second condition. It specifies that you want to find documents where the field named "fieldName2" matches the specified condition ("condition2").

**Example**

[

{

"Name":"Sam",

"hobbies":["Painting","Reading","Writing"],

"Age":45

},

{

"Name":"Somu",

"hobbies":["Walking","Reading","Writing"],

"Age":45

},

{

"Name":"Raju",

"hobbies":["Painting","Reading","Cricket"],

"Age":45

}

]

db.std.find({

$and:[

{"hobbies" : "Painting"},

{"hobbies" : "Reading"}

]

})

The MongoDB query you provided searches for documents in the "std" collection where the "hobbies" array contains both "Painting" and "Reading."

1. **db.std.find(...):**This part of the code performs a find operation in the "std" collection.
2. **$and:** The $and operator is used to specify that all the conditions inside it must be true for a document to be considered a match. In this case, it's used to combine two conditions.
3. **{"hobbies": "Painting"}:** This is the first condition. It specifies that you want to find documents where the "hobbies" array contains the element "Painting."
4. **{"hobbies": "Reading"}:**This is the second condition. It specifies that you want to find documents where the "hobbies" array contains the element "Reading."

The query effectively looks for documents where both conditions within the $and operator are satisfied. In this case, it's searching for documents where the "hobbies" array contains both "Painting" and "Reading."

#### Output

