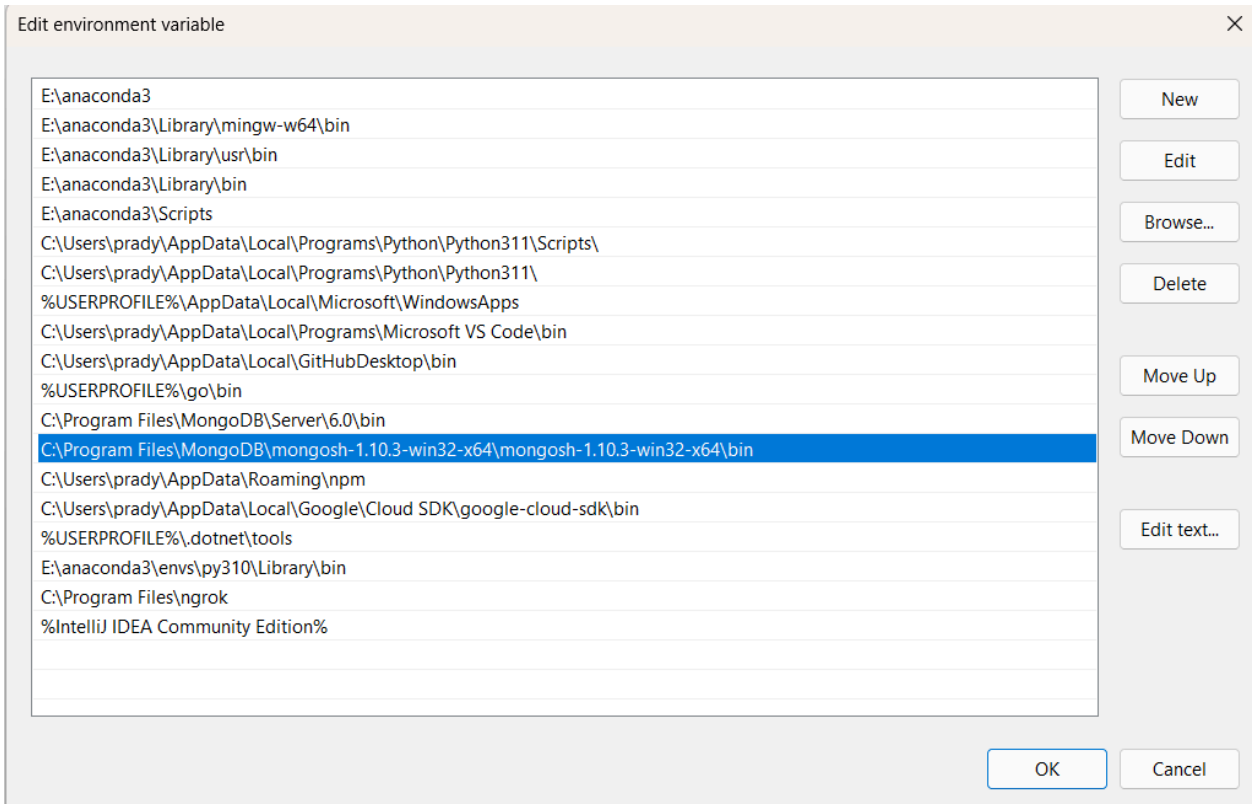
1. MongoDB Installation and Configuration in Windows

Step 1: Download MongoDB
1. Visit: https://www.mongodb.com/try/download/community
2. Download the **Windows (x64) MSI** package.

Step 2: Install MongoDB
1. Open the downloaded `.msi` file and follow the installation wizard.
2. Select **Complete Installation** and ensure MongoDB is set to run as a service.
3. Click **Install**, then **Finish** after completion.

Step 3: Configure Environment Variables
1. Add `C:\Program Files\MongoDB\Server\<version>\Bin` to **System PATH**.



Step 4: Configure the data & log folder and set MongoDB as service by typing this command.
Note: This is a one line command.

```
mongod --directoryperdb --dbpath "C:\Program Files\MongoDB\data\db"
--logpath "C:\Program Files\MongoDB\log\mongo.log" --logappend --rest --install
```

Step 5: Verify Installation
```
mongod --version
```

```
PS C:\Users\prady> mongod --version
db version v6.0.8
Build Info: {
    "version": "6.0.8",
    "gitVersion": "3d84c0dd4e5d99be0d69003652313e7eaf4cdd74",
    "modules": [],
    "allocator": "tcmalloc",
    "environment": {
        "distmod": "windows",
        "distarch": "x86_64",
        "target_arch": "x86_64"
    }
}
```

Step 5: Start MongoDB Service
Run the following command as Administrator:
```sh
net start MongoDB
```

```
C:\Windows\system32>net start MongoDB
The MongoDB service is starting..
The MongoDB service was started successfully.


C:\Windows\system32>net stop MongoDB
The MongoDB service is stopping.
The MongoDB service was stopped successfully.
```

Step 6: Connect to MongoDB Shell by running this command
```
mongosh
```

```
PS C:\Users\prady> mongosh
Current Mongosh Log ID: 67b62dfe2e97ac587e0c96a9
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1
.10.3
Using MongoDB:          6.0.8
Using Mongosh:          1.10.3

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

------
   The server generated these startup warnings when booting
   2025-02-19T22:49:36.811+05:30: Access control is not enabled for the database. Read and write access to data and conf
iguration is unrestricted
------

test>
```

2. To create and drop a database in MongoDB

To create and drop a database in MongoDB, follow these step-by-step instructions:

---

1. Open MongoDB Shell

Ensure MongoDB is installed and running on your system. Open Command Prompt (cmd) or Terminal and type:

mongosh

This will start the MongoDB shell.

---

2. Check Existing Databases

To see a list of all existing databases, run:

show dbs

```
test> show dbs
IS4B    144.00 KiB
admin    40.00 KiB
config   72.00 KiB
local   360.00 KiB
test     60.00 KiB
test> |
```

This will display all available databases.

---

3. Create a New Database

In MongoDB, a database is created when you switch to a non-existing database and insert data into it.

To create (or switch to) a new database, use:

use myDatabase

```
test> use myDatabase
switched to db myDatabase
myDatabase>
```

Here, `myDatabase` is the name of the database you want to create.

---

4. Insert Data into the Database

Since MongoDB does not create an empty database, you must insert at least one document into a collection.

Create a collection and insert a document:

db.users.insertOne({ name: "John Doe", age: 25 })

```
myDatabase> db.users.insertOne({ name: "John Doe", age: 25 })
{
  acknowledged: true,
  insertedId: ObjectId("67b630e115dbbba8e86820b6")
}
myDatabase>
```

Here:

- `users` is the collection name.
- The document `{ name: "John Doe", age: 25 }` is inserted.

Now, the database `myDatabase` is officially created.

---

5. Verify Database Creation

Run:

show dbs

```
myDatabase> show dbs
IS4B        144.00 KiB
admin        40.00 KiB
config       72.00 KiB
local       360.00 KiB
myDatabase   40.00 KiB
test         60.00 KiB
myDatabase>
```

If you don't see `myDatabase`, switch to it using:

use myDatabase

Then, check collections:

show collections

```
myDatabase> show collections
users
myDatabase>
```

You should see the `users` collection.

---

6. Drop a Database

To delete (drop) a database, first switch to it:

use myDatabase

Then, drop it with:

db.dropDatabase()

```
myDatabase> db.dropDatabase()
{ ok: 1, dropped: 'myDatabase' }
myDatabase>
```

This will remove `myDatabase` permanently.

---

7. Verify Deletion

Run:

show dbs

```
myDatabase> show dbs
IS4B    144.00 KiB
admin    40.00 KiB
config  108.00 KiB
local   360.00 KiB
test     60.00 KiB
```

You should not see `myDatabase` in the list.

3. To create a collection in MongoDB, follow these step-by-step instructions:

---

1. Open MongoDB Shell

Ensure MongoDB is installed and running. Open Command Prompt (cmd) or Terminal and type:

mongosh

This starts the MongoDB shell.

---

2. Select a Database

Before creating a collection, you must choose a database. If the database doesn't exist, MongoDB will create it when you insert data.

use myDatabase

```
test> use myDatabase
switched to db myDatabase
myDatabase>
```

Here, `myDatabase` is the name of the database where you want to create a collection.

---

3. Create a Collection Explicitly (Optional)

MongoDB automatically creates a collection when you insert data into it. However, you can also manually create a collection using:

db.createCollection("admin")

```
myDatabase> db.createCollection("admin")
{ ok: 1 }
```

This creates a collection named `users`.

You can also specify options like `capped`, `size`, and `max`:

db.createCollection("logs", { capped: true, size: 10000, max: 500 })

- `capped: true` → The collection has a fixed size.
- `size: 10000` → The collection's maximum size (in bytes).
- `max: 500` → Limits the number of documents.

---

4. Verify Collection Creation

To check the collections in the database, run:

show collections

```
myDatabase> show collections
admin
users
myDatabase>
```

It will list all collections, including `admin` and `logs`.

---

5. Create a Collection by Inserting Data (Implicit Method)

Instead of manually creating a collection, you can simply insert a document, and MongoDB will create the collection automatically:

db.products.insertOne({ name: "Laptop", price: 75000 })

```
myDatabase> db.products.insertOne({ name: "Laptop", price: 75000 })
{
  acknowledged: true,
  insertedId: ObjectId("67b632bc72216bfa97548fb6")
}
myDatabase>
```

Now, the `products` collection is created.

---

6. Verify the Collection

Run:

show collections

```
myDatabase> show collections
admin
products
users
myDatabase>
```

You should see `products` along with other collections.

4. To create a collection with options before inserting documents and drop the collection, follow these step-by-step instructions:

---

1. Open MongoDB Shell

Ensure MongoDB is installed and running. Open **Command Prompt (cmd) or Terminal** and type:

mongosh

This starts the MongoDB shell.

---

2. Select or Create a Database

Before creating a collection, you must select a database:

use myDatabase

```
test> use myDatabase
switched to db myDatabase
myDatabase>
```

If `myDatabase` does not exist, MongoDB will create it when a document is inserted.

---

3. Create a Collection with Options

Instead of allowing MongoDB to automatically create a collection, you can explicitly create one with options.

Example 1: Creating a Capped Collection

A capped collection is a fixed-size collection where old documents are removed when the limit is reached.

db.createCollection("logs", { capped: true, size: 5000, max: 100 })

```
myDatabase> db.createCollection("logs", { capped: true, size: 5000, max: 100 })
{ ok: 1 }
myDatabase>
```

- `capped: true` → Enables capped collection.
- `size: 5000` → Sets the maximum size in bytes.
- `max: 100` → Limits the number of documents.

Example 2: Creating a Collection with Validation Rules

You can define a schema using validation rules:

```
db.createCollection("students", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "age"],
      properties: {
        name: {
          bsonType: "string",
          description: "Name must be a string"
        },
        age: {
          bsonType: "int",
          minimum: 18,
          description: "Age must be an integer and at least 18"
        }
      }
    }
  }
})
```

```
myDatabase> db.createCollection("students", {
...     validator: {
...        $jsonSchema: {
...           bsonType: "object",
...           required: ["name", "age"],
...           properties: {
...              name: {
...                 bsonType: "string",
...                 description: "Name must be a string"
...              },
...              age: {
...                 bsonType: "int",
...                 minimum: 18,
...                 description: "Age must be an integer and at least 18"
...              }
...           }
...        }
...     }
... })
{ ok: 1 }
myDatabase>
```

- This ensures `name` is a string and `age` is an integer of at least 18.

---

4. Insert Documents into the Collection

Once the collection is created, you can insert data:

db.students.insertOne({ name: "John Doe", age: 25 })

```
myDatabase> db.students.insertOne({ name: "John Doe", age: 25 })
{
  acknowledged: true,
  insertedId: ObjectId("67b6347972216bfa97548fb7")
}
myDatabase>
```

For multiple documents:

db.students.insertMany([
** { name: "Alice", age: 22 },**
** { name: "Bob", age: 30 }**
])

```
myDatabase> db.students.insertMany([
...     { name: "Alice", age: 22 },
...     { name: "Bob", age: 30 }
... ])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("67b6349772216bfa97548fb8"),
    '1': ObjectId("67b6349772216bfa97548fb9")
  }
}
myDatabase>
```

---

5. Verify the Collection

To check all collections in the database:

show collections

```
myDatabase> show collections
admin
logs
products
students
users
myDatabase>
```

---

6. Drop the Collection

If you need to remove a collection, use:

db.students.drop()

```
myDatabase> db.students.drop()
true
```

This permanently deletes the `students` collection.

To verify, run:

show collections

```
myDatabase> show collections
admin
logs
products
users
myDatabase>
```

The collection should no longer be listed.

---

5. To **insert a document into a MongoDB collection**, follow these steps:

---

1. Open MongoDB Shell

Ensure MongoDB is installed and running. Open **Command Prompt (cmd) or Terminal** and type:

mongosh

This starts the MongoDB shell.

---

2. Select the Database

Before inserting a document, switch to the database where you want to insert it:

use myDatabase

```
test> use myDatabase
switched to db myDatabase
myDatabase>
```

If `myDatabase` does not exist, it will be created when a document is inserted.

5(a): Insert a Single Document

To insert a single document into a collection, use:

db.users.insertOne({ name: "John Doe", age: 25, city: "Mangalore" })

```
myDatabase> db.users.insertOne({ name: "John Doe", age: 25, city: "Mangalore" })
{
  acknowledged: true,
  insertedId: ObjectId("67b6356f72216bfa97548fba")
}
myDatabase>
```

- `users` is the collection name.
- The inserted document has fields `name`, `age`, and `city`.

Verify Insertion

To check if the document was inserted:

db.users.find().pretty()

```
myDatabase> db.users.find().pretty()
[
  {
    _id: ObjectId("67b6356f72216bfa97548fba"),
    name: 'John Doe',
    age: 25,
    city: 'Mangalore'
  }
]
myDatabase>
```

This will display the document in a readable format.

5(b): Insert Multiple Documents

To insert multiple documents at once, use:

db.users.insertMany([
{ name: "Alice", age: 22, city: "Bangalore" },
{ name: "Bob", age: 30, city: "Delhi" },
{ name: "Charlie", age: 27, city: "Mumbai" }
])

```
myDatabase> db.users.insertMany([
...     { name: "Alice", age: 22, city: "Bangalore" },
...     { name: "Bob", age: 30, city: "Delhi" },
...     { name: "Charlie", age: 27, city: "Mumbai" }
... ])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("67b6358872216bfa97548fbb"),
    '1': ObjectId("67b6358872216bfa97548fbc"),
    '2': ObjectId("67b6358872216bfa97548fbd")
  }
}
myDatabase>
```

Verify Multiple Documents

To display all inserted documents:

db.users.find().pretty()

```
myDatabase> db.users.find().pretty()
[
  {
    _id: ObjectId("67b6356f72216bfa97548fba"),
    name: 'John Doe',
    age: 25,
    city: 'Mangalore'
  },
  {
    _id: ObjectId("67b6358872216bfa97548fbb"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore'
  },
  {
    _id: ObjectId("67b6358872216bfa97548fbc"),
    name: 'Bob',
    age: 30,
    city: 'Delhi'
  },
  {
    _id: ObjectId("67b6358872216bfa97548fbd"),
    name: 'Charlie',
    age: 27,
    city: 'Mumbai'
  }
]
myDatabase>
```

Highlights

- `insertOne()` → Inserts a single document.
- `insertMany()` → Inserts multiple documents.
- `find().pretty()` → Displays inserted documents in a readable format.

6. To **query all documents in JSON format and query based on criteria** in MongoDB, follow these steps:

---

1. Open MongoDB Shell

Ensure MongoDB is installed and running. Open **Command Prompt (cmd) or Terminal** and type:

mongosh

This starts the MongoDB shell.

---

2. Select the Database

Switch to the database where your collection exists:

use myDatabase

```
test> use myDatabase
switched to db myDatabase
myDatabase>
```

---

3. Query All Documents in JSON Format

Before retrieving populate the collection by insertMany

To insert multiple documents at once, use:

```
db.users.insertMany([
  { name: "John Doe", age: 25, city: "Mangalore" },
  { name: "Alice", age: 22, city: "Bangalore" },
  { name: "Bob", age: 30, city: "Delhi" },
  { name: "Charlie", age: 27, city: "Mumbai" }
])
```

To retrieve all documents from a collection in JSON format, use:

db.users.find().pretty()

```
myDatabase> db.users.find().pretty()
[
  {
    _id: ObjectId("67b6356f72216bfa97548fba"),
    name: 'John Doe',
    age: 25,
    city: 'Mangalore'
  },
  {
    _id: ObjectId("67b6358872216bfa97548fbb"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore'
  },
  {
    _id: ObjectId("67b6358872216bfa97548fbc"),
    name: 'Bob',
    age: 30,
    city: 'Delhi'
  },
  {
    _id: ObjectId("67b6358872216bfa97548fbd"),
    name: 'Charlie',
    age: 27,
    city: 'Mumbai'
  }
]
myDatabase>
```

- `.find()` retrieves all documents.
- `.pretty()` formats the output in a readable JSON format.

---

4. Query Documents Based on Criteria

You can filter documents based on specific fields.

a) Query by Exact Match

Find users from a specific city:

db.users.find({ city: "Bangalore" }).pretty()

```
myDatabase> db.users.find({ city: "Bangalore" }).pretty()
[
  {
    _id: ObjectId("67b6358872216bfa97548fbb"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore'
  }
]
myDatabase>
```

This returns all users whose `city` is **Bangalore**.

b) Query Using Comparison Operators

Find users older than 25:

db.users.find({ age: { $gt: 25 } }).pretty()

```
myDatabase> db.users.find({ age: { $gt: 25 } }).pretty()
[
  {
    _id: ObjectId("67b6358872216bfa97548fbc"),
    name: 'Bob',
    age: 30,
    city: 'Delhi'
  },
  {
    _id: ObjectId("67b6358872216bfa97548fbd"),
    name: 'Charlie',
    age: 27,
    city: 'Mumbai'
  }
]
myDatabase>
```

- `$gt` → Greater than
- `$lt` → Less than
- `$gte` → Greater than or equal to
- `$lte` → Less than or equal to

c) Query with Logical Operators

Find users who live in **Bangalore** or **Mumbai**:

db.users.find({ $or: [{ city: "Bangalore" }, { city: "Mumbai" }] }).pretty()

```
myDatabase> db.users.find({ $or: [{ city: "Bangalore" }, { city: "Mumbai" }] }).pretty()
[
  {
    _id: ObjectId("67b6358872216bfa97548fbb"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore'
  },
  {
    _id: ObjectId("67b6358872216bfa97548fbd"),
    name: 'Charlie',
    age: 27,
    city: 'Mumbai'
  }
]
myDatabase>
```

- `$or` → Matches documents where at least one condition is true.

d) Query with Specific Fields

Return only `name` and `age` fields, excluding `_id`:

db.users.find({}, { _id: 0, name: 1, age: 1 }).pretty()

```
myDatabase> db.users.find({}, { _id: 0, name: 1, age: 1 }).pretty()
[
  { name: 'John Doe', age: 25 },
  { name: 'Alice', age: 22 },
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 27 }
]
myDatabase>
```

- `_id: 0` → Excludes the `_id` field.
- `name: 1, age: 1` → Includes only `name` and `age`.

e) Query with Regex (Pattern Matching)

Find users whose names start with **"A"**:

db.users.find({ name: /^A/ }).pretty()

```
myDatabase> db.users.find({ name: /^A/ }).pretty()
[
  {
    _id: ObjectId("67b6358872216bfa97548fbb"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore'
  }
]
myDatabase>
```

- Uses a **regular expression** to match names starting with `A`.

f) Query with Sorting

Sort users by age in **descending** order:

db.users.find().sort({ age: -1 }).pretty()

```
myDatabase> db.users.find().sort({ age: -1 }).pretty()
[
  {
    _id: ObjectId("67b6358872216bfa97548fbc"),
    name: 'Bob',
    age: 30,
    city: 'Delhi'
  },
  {
    _id: ObjectId("67b6358872216bfa97548fbd"),
    name: 'Charlie',
    age: 27,
    city: 'Mumbai'
  },
  {
    _id: ObjectId("67b6356f72216bfa97548fba"),
    name: 'John Doe',
    age: 25,
    city: 'Mangalore'
  },
  {
    _id: ObjectId("67b6358872216bfa97548fbb"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore'
  }
]
myDatabase>
```

- 1 → Ascending order.
- -1 → Descending order.

g) Query with Limit and Skip

Retrieve only **2 documents**, skipping the first **1**:

db.users.find().skip(1).limit(2).pretty()

```
myDatabase> db.users.find().skip(1).limit(2).pretty()
[
  {
    _id: ObjectId("67b6358872216bfa97548fbb"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore'
  },
  {
    _id: ObjectId("67b6358872216bfa97548fbc"),
    name: 'Bob',
    age: 30,
    city: 'Delhi'
  }
]
myDatabase>
```

---

7. To **update a document in MongoDB using the `updateOne()` and `save()` methods**, follow these steps:

---

1. Open MongoDB Shell

Ensure MongoDB is installed and running. Open **Command Prompt (cmd) or Terminal** and type:

mongosh

This starts the MongoDB shell.

---

2. Select the Database

Switch to the database where your collection exists:

use myDatabase

```
test> use myDatabase
switched to db myDatabase
myDatabase>
```

---

3. View Existing Documents

Before updating, check existing documents:

db.users.find().pretty()

```
myDatabase> db.users.find().pretty()
[
  {
    _id: ObjectId("67b6356f72216bfa97548fba"),
    name: 'John Doe',
    age: 25,
    city: 'Mangalore'
  },
  {
    _id: ObjectId("67b6358872216bfa97548fbb"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore'
  },
  {
    _id: ObjectId("67b6358872216bfa97548fbc"),
    name: 'Bob',
    age: 30,
    city: 'Delhi'
  },
  {
    _id: ObjectId("67b6358872216bfa97548fbd"),
    name: 'Charlie',
    age: 27,
    city: 'Mumbai'
  }
]
myDatabase>
```

This will display all documents in the `users` collection.

7(a): Using `updateOne()` Method

The `updateOne()` method updates the **first matching document**.

Example: Update a User's Age

db.users.updateOne(
** { name: "John Doe" }, // Search criteria**
** { $set: { age: 30 } } // Update operation**
)

```
myDatabase> db.users.updateOne(
...     { name: "John Doe" },   // Search criteria
...     { $set: { age: 30 } }   // Update operation
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
myDatabase>
```

- Finds the first document where `name` is **"John Doe"**.
- Updates the `age` field to **30**.

Example: Add a New Field

db.users.updateOne(
** { name: "Alice" },**
** { $set: { city: "Mangalore" } }**
)

```
myDatabase> db.users.updateOne(
...     { name: "Alice" },
...     { $set: { city: "Mangalore" } }
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
myDatabase>
```

If `city` does not exist, it will be added.

Verify the Update

db.users.find({ name: "John Doe" }).pretty()

```
myDatabase> db.users.find({ name: "John Doe" }).pretty()
[
  {
    _id: ObjectId("67b6356f72216bfa97548fba"),
    name: 'John Doe',
    age: 30,
    city: 'Mangalore'
  }
]
```

7(b): Using `replaceOne()` Method

The `replaceOne()` method replaces an entire document. This is rarely used now, as `updateOne()` is preferred.

Example: Replace Entire Document

db.users.replaceOne(
{ _id: ObjectId("67b6356f72216bfa97548fba") },
{ name: "John Doe", age: 35, city: "Mumbai" }
)

```
myDatabase> db.users.replaceOne( { _id: ObjectId("67b6356f72216bfa97548fba") }, { name: "John Doe", age: 35, city: "Mumb
ai" } )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

- You must include the **`_id`** field, or MongoDB will treat it as a new document.
- If `John Doe` already existed, it is **completely replaced** by this new document.

Verify the Change

db.users.find().pretty()

```
myDatabase>  db.users.find({ name: "John Doe" }).pretty()
[
  {
    _id: ObjectId("67b6356f72216bfa97548fba"),
    name: 'John Doe',
    age: 35,
    city: 'Mumbai'
  }
]
myDatabase>
```

---

8. **delete documents using the latest MongoDB methods (`deleteOne()` and `deleteMany()`).**

---

1. Open MongoDB Shell

Ensure MongoDB is installed and running. Open **Command Prompt (cmd) or Terminal** and type:

mongosh

This starts the MongoDB shell.

---

2. Select the Database

Switch to the database where your collection exists:

use myDatabase

```
test> use myDatabase
switched to db myDatabase
myDatabase>
```

---

3. View Existing Documents

Before deleting, check existing documents and if empty insert

To insert multiple documents at once, use:

```
db.users.insertMany([
  { name: "John Doe", age: 25, city: "Mangalore" },
  { name: "Alice", age: 22, city: "Bangalore" },
  { name: "Bob", age: 30, city: "Delhi" },
  { name: "Charlie", age: 27, city: "Bangalore" }
])

db.users.find().pretty()
```

```
myDatabase> db.users.find().pretty()
[
  {
    _id: ObjectId("67b63f4d72216bfa97548fbe"),
    name: 'John Doe',
    age: 25,
    city: 'Mangalore'
  },
  {
    _id: ObjectId("67b63f4d72216bfa97548fbf"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore'
  },
  {
    _id: ObjectId("67b63f4d72216bfa97548fc0"),
    name: 'Bob',
    age: 30,
    city: 'Delhi'
  },
  {
    _id: ObjectId("67b63f4d72216bfa97548fc1"),
    name: 'Charlie',
    age: 27,
    city: 'Bangalore'
  }
]
myDatabase>
```

8(a): Delete a Document Using `deleteOne()`

`deleteOne()` removes **only the first document** that matches the given condition.

Example: Delete a User Named "John Doe"

db.users.deleteOne({ name: "John Doe" })

```
myDatabase> db.users.deleteOne({ name: "John Doe" })
{ acknowledged: true, deletedCount: 1 }
myDatabase>
```

- This deletes **only one document** where `name` is **"John Doe"**.

Verify Deletion

db.users.find({ name: "John Doe" }).pretty()

```
myDatabase> db.users.find({ name: "John Doe" }).pretty()

myDatabase>
```

If no documents appear, the deletion was successful.

8(b): Remove All Documents Matching a Condition

To delete **all documents** that match a condition, use `deleteMany()`.

Example: Delete All Users from "Bangalore"

db.users.deleteMany({ city: "Bangalore" })

```
myDatabase> db.users.deleteMany({ city: "Bangalore" })
{ acknowledged: true, deletedCount: 2 }
myDatabase>
```

- This deletes **all users** whose `city` is **Bangalore**.

Verify Deletion

db.users.find({ city: "Bangalore" }).pretty()

```
myDatabase> db.users.find({ city: "Bangalore" }).pretty()

myDatabase>
```

8(c): Remove All Documents in a Collection

If you want to delete **all documents** but keep the collection structure:

db.users.deleteMany({})

```
myDatabase> db.users.deleteMany({})
{ acknowledged: true, deletedCount: 2 }
myDatabase>
```

- This **removes all documents** in the `users` collection.

If you want to **drop the entire collection**, use:

db.users.drop()

```
myDatabase> db.users.drop()
true
myDatabase>
```

- This **removes the collection itself**.

```
myDatabase> show collections
admin
logs
products
myDatabase>
```

9. MongoDB Projection

Projection in MongoDB is used to **select specific fields** from documents instead of retrieving the entire document. This helps **optimize performance** by fetching only necessary data.

---

1. Open MongoDB Shell

Ensure MongoDB is installed and running. Open **Command Prompt (cmd) or Terminal** and type:

mongosh

This starts the MongoDB shell.

---

2. Select the Database

Switch to the database where your collection exists:

use myDatabase

```
test> use myDatabase
switched to db myDatabase
myDatabase>
```

---

3. View Existing Documents

To insert multiple documents at once, use:

```
db.users.insertMany([
  { name: "John Doe", age: 25, city: "Mangalore", email: "john.doe@example.com" },
  { name: "Alice", age: 22, city: "Bangalore", email: "alice@example.com" },
  { name: "Bob", age: 30, city: "Delhi", email: "bob@example.com" },
  { name: "Charlie", age: 27, city: "Bangalore", email: "charlie@example.com" }
])
```

Before applying projection, check the existing documents:

db.users.find().pretty()

```
myDatabase> db.users.find().pretty()
[
  {
    _id: ObjectId("67b6405472216bfa97548fc2"),
    name: 'John Doe',
    age: 25,
    city: 'Mangalore'
  },
  {
    _id: ObjectId("67b6405472216bfa97548fc3"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore'
  },
  {
    _id: ObjectId("67b6405472216bfa97548fc4"),
    name: 'Bob',
    age: 30,
    city: 'Delhi'
  },
  {
    _id: ObjectId("67b6405472216bfa97548fc5"),
    name: 'Charlie',
    age: 27,
    city: 'Bangalore'
  }
]
myDatabase>
```

9(a): Projection to Include Specific Fields

To fetch only **name** and **age**, excluding `_id`, use:

db.users.find({}, { name: 1, age: 1, _id: 0 }).pretty()

```
myDatabase> db.users.find({}, { name: 1, age: 1, _id: 0 }).pretty()
[
  { name: 'John Doe', age: 25 },
  { name: 'Alice', age: 22 },
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 27 }
]
myDatabase>
```

- `name: 1, age: 1` → Includes only these fields.
- `_id: 0` → Excludes `_id` from the result

9(b): Exclude a Specific Field

To **exclude** the `email` field but keep everything else:

db.users.find({}, { email: 0 }).pretty()

```
myDatabase> db.users.find({}, { email: 0 }).pretty()
[
  {
    _id: ObjectId("67b640ef72216bfa97548fc6"),
    name: 'John Doe',
    age: 25,
    city: 'Mangalore'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc7"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc8"),
    name: 'Bob',
    age: 30,
    city: 'Delhi'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc9"),
    name: 'Charlie',
    age: 27,
    city: 'Bangalore'
  }
]
myDatabase>
```

- This will return all fields **except** `email`.

9(c): Projection with Query Condition

To get only `name` and `age` for users from **Mangalore**:

db.users.find({ city: "Mangalore" }, { name: 1, age: 1, _id: 0 }).pretty()

```
myDatabase> db.users.find({ city: "Mangalore" }, { name: 1, age: 1, _id: 0 }).pretty()
[ { name: 'John Doe', age: 25 } ]
myDatabase>
```

9(d): Using Projection with Sorting

To retrieve only `name` and `age`, sorted by age in **descending** order:

db.users.find({}, { name: 1, age: 1, _id: 0 }).sort({ age: -1 }).pretty()

```
myDatabase> db.users.find({}, { name: 1, age: 1, _id: 0 }).sort({ age: -1 }).pretty()
[
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 27 },
  { name: 'John Doe', age: 25 },
  { name: 'Alice', age: 22 }
]
myDatabase>
```

---

10. Limit, Skip, and Sort Methods in MongoDB

MongoDB provides `limit()`, `skip()`, and `sort()` methods to control and organize query results efficiently.

---

1. Open MongoDB Shell

Ensure MongoDB is installed and running. Open **Command Prompt (cmd) or Terminal** and type:

mongosh

This starts the MongoDB shell.

---

2. Select the Database

Switch to the database where your collection exists:

use myDatabase

```
test> use myDatabase
switched to db myDatabase
myDatabase>
```

---

3. View Existing Documents

Insert multiple documents first at once, use:

```
db.users.insertMany([
  { name: "John Doe", age: 25, city: "Mangalore", email: "john.doe@example.com" },
  { name: "Alice", age: 22, city: "Bangalore", email: "alice@example.com" },
  { name: "Bob", age: 30, city: "Delhi", email: "bob@example.com" },
  { name: "Charlie", age: 27, city: "Bangalore", email: "charlie@example.com" }
])
```

Before applying **limit, skip, and sort**, check the existing documents:

db.users.find().pretty()

```
myDatabase> db.users.find().pretty()
[
  {
    _id: ObjectId("67b640ef72216bfa97548fc6"),
    name: 'John Doe',
    age: 25,
    city: 'Mangalore',
    email: 'john.doe@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc7"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore',
    email: 'alice@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc8"),
    name: 'Bob',
    age: 30,
    city: 'Delhi',
    email: 'bob@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc9"),
    name: 'Charlie',
    age: 27,
    city: 'Bangalore',
    email: 'charlie@example.com'
  }
]
```

10(a): Using `limit()`

The `limit(n)` method restricts the number of documents returned.

Example: Retrieve Only 2 Documents

db.users.find().limit(2).pretty()

```
myDatabase> db.users.find().limit(2).pretty()
[
  {
    _id: ObjectId("67b640ef72216bfa97548fc6"),
    name: 'John Doe',
    age: 25,
    city: 'Mangalore',
    email: 'john.doe@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc7"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore',
    email: 'alice@example.com'
  }
]
myDatabase> |
```

- Returns **only the first 2 documents** in the query result.

10(b): Using `skip()`

The `skip(n)` method **skips** a specified number of documents from the result.

Example: Skip the First 2 Documents and Retrieve the Next 2

db.users.find().skip(2).limit(2).pretty()

```
myDatabase> db.users.find().skip(2).limit(2).pretty()
[
  {
    _id: ObjectId("67b640ef72216bfa97548fc8"),
    name: 'Bob',
    age: 30,
    city: 'Delhi',
    email: 'bob@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc9"),
    name: 'Charlie',
    age: 27,
    city: 'Bangalore',
    email: 'charlie@example.com'
  }
]
myDatabase>
```

- Skips **first 2 documents** and returns the **next 2**.

10(c): Using `sort()`

The `sort({ field: order })` method **sorts** documents based on a field.

- **Ascending order:** `1`
- **Descending order:** `-1`

Example: Sort Users by Age (Ascending)

db.users.find().sort({ age: 1 }).pretty()

```
myDatabase> db.users.find().sort({ age: 1 }).pretty()
[
  {
    _id: ObjectId("67b640ef72216bfa97548fc7"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore',
    email: 'alice@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc6"),
    name: 'John Doe',
    age: 25,
    city: 'Mangalore',
    email: 'john.doe@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc9"),
    name: 'Charlie',
    age: 27,
    city: 'Bangalore',
    email: 'charlie@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc8"),
    name: 'Bob',
    age: 30,
    city: 'Delhi',
    email: 'bob@example.com'
  }
]
myDatabase>
```

- Sorts users **from youngest to oldest**.

Example: Sort Users by Age (Descending)

```
myDatabase> db.users.find().sort({ age: -1 }).pretty()
[
  {
    _id: ObjectId("67b640ef72216bfa97548fc8"),
    name: 'Bob',
    age: 30,
    city: 'Delhi',
    email: 'bob@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc9"),
    name: 'Charlie',
    age: 27,
    city: 'Bangalore',
    email: 'charlie@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc6"),
    name: 'John Doe',
    age: 25,
    city: 'Mangalore',
    email: 'john.doe@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc7"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore',
    email: 'alice@example.com'
  }
]
myDatabase>
```

- **db.users.find().sort({ age: -1 }).pretty()**
- Sorts users **from oldest to youngest**.

Example: Sort by Multiple Fields

Sort first by `city` (A-Z) and then by `age` (ascending):

db.users.find().sort({ city: 1, age: 1 }).pretty()

```
myDatabase> db.users.find().sort({ city: 1, age: 1 }).pretty()
[
  {
    _id: ObjectId("67b640ef72216bfa97548fc7"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore',
    email: 'alice@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc9"),
    name: 'Charlie',
    age: 27,
    city: 'Bangalore',
    email: 'charlie@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc8"),
    name: 'Bob',
    age: 30,
    city: 'Delhi',
    email: 'bob@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc6"),
    name: 'John Doe',
    age: 25,
    city: 'Mangalore',
    email: 'john.doe@example.com'
  }
]
myDatabase>
```

10(d): Combining `limit()`, `skip()`, and `sort()`

You can **combine** all three for **pagination**.

Example: Get the 2nd Page of Results (2 per page), Sorted by Age Descending

db.users.find().sort({ age: -1 }).skip(2).limit(2).pretty()

```
myDatabase> db.users.find().sort({ age: -1 }).skip(2).limit(2).pretty()
[
  {
    _id: ObjectId("67b640ef72216bfa97548fc6"),
    name: 'John Doe',
    age: 25,
    city: 'Mangalore',
    email: 'john.doe@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc7"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore',
    email: 'alice@example.com'
  }
]
myDatabase>
```

- Sorts by **age in descending order**.
- Skips the **first 2 documents**.
- Returns **the next 2 documents**.

---

11. MongoDB Indexing

Indexing in MongoDB helps optimize query performance by allowing faster data retrieval. Without indexes, MongoDB performs a **collection scan**, which is slow for large datasets.

---

1. Open MongoDB Shell

Ensure MongoDB is installed and running. Open **Command Prompt (cmd) or Terminal** and type:

mongosh

This starts the MongoDB shell.

---

2. Select the Database

Switch to the database where your collection exists:

use myDatabase

```
test> use myDatabase
switched to db myDatabase
myDatabase>
```

---

3. View Existing Documents

Insert multiple documents first at once, use:

```
db.users.insertMany([
  { name: "John Doe", age: 25, city: "Mangalore", email: "john.doe@example.com" },
  { name: "Alice", age: 22, city: "Bangalore", email: "alice@example.com" },
  { name: "Bob", age: 30, city: "Delhi", email: "bob@example.com" },
  { name: "Charlie", age: 27, city: "Bangalore", email: "charlie@example.com" }
])
```

Before applying indexing, check the existing documents:

db.users.find().pretty()

```
myDatabase> db.users.find().pretty()
[
  {
    _id: ObjectId("67b640ef72216bfa97548fc6"),
    name: 'John Doe',
    age: 25,
    city: 'Mangalore',
    email: 'john.doe@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc7"),
    name: 'Alice',
    age: 22,
    city: 'Bangalore',
    email: 'alice@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc8"),
    name: 'Bob',
    age: 30,
    city: 'Delhi',
    email: 'bob@example.com'
  },
  {
    _id: ObjectId("67b640ef72216bfa97548fc9"),
    name: 'Charlie',
    age: 27,
    city: 'Bangalore',
    email: 'charlie@example.com'
  }
]
myDatabase>
```

11(a): Create an Index

Indexes improve query performance. To create an index on a field, use:

Create an Index on a Single Field

db.users.createIndex({ name: 1 })

```
myDatabase> db.users.createIndex({ name: 1 })
name_1
myDatabase>
```

- Creates an **ascending index** on the `name` field.
- `1` → Ascending order.
- `-1` → Descending order.

Create a Compound Index (Multiple Fields)

db.users.createIndex({ city: 1, age: -1 })

```
myDatabase> db.users.createIndex({ city: 1, age: -1 })
city_1_age_-1
myDatabase>
```

- Sorts by `city` in **ascending** order.
- If `city` is the same, sorts by `age` in **descending** order.

Create a Unique Index

db.users.createIndex({ email: 1 }, { unique: true })

```
myDatabase> db.users.createIndex({ email: 1 }, { unique: true })
email_1
myDatabase>
```

- Ensures that `email` values **must be unique**.

11(b): Find Existing Indexes

To check all indexes on a collection:

db.users.getIndexes()

```
myDatabase> db.users.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { name: 1 }, name: 'name_1' },
  { v: 2, key: { city: 1, age: -1 }, name: 'city_1_age_-1' },
  { v: 2, key: { email: 1 }, name: 'email_1', unique: true }
]
myDatabase>
```

11(c): Drop an Index

To remove a specific index:

db.users.dropIndex("name_1")

```
myDatabase> db.users.dropIndex("name_1")
{ nIndexesWas: 4, ok: 1 }
myDatabase>
```

- `"name_1"` is the index name assigned automatically.

To drop an index on multiple fields:

db.users.dropIndex({ city: 1, age: -1 })

```
myDatabase> db.users.dropIndex({ city: 1, age: -1 })
{ nIndexesWas: 3, ok: 1 }
myDatabase>
```

11(d): Drop All Indexes

To remove **all indexes** except the default `_id` index:

db.users.dropIndexes()

```
myDatabase> db.users.dropIndexes()
{
  nIndexesWas: 2,
  msg: 'non-_id indexes dropped for collection',
  ok: 1
}
myDatabase>
```

Testing Index Performance in MongoDB (Proof of Improvement) (Appendix)

To demonstrate that **indexes improve performance**, we will use the `explain("executionStats")` method to compare query execution **before and after indexing**.

1. Insert a Large Number of Documents (for Testing)

If your collection has **only a few documents**, the performance difference may not be noticeable. Let's add **10,000 users** for proper testing:

```
for (let i = 0; i < 10000; i++) {
  db.users.insertOne({
    name: "User" + i,
    age: Math.floor(Math.random() * 50) + 20,
    city: ["Mangalore", "Bangalore", "Delhi", "Mumbai"][Math.floor(Math.random() * 4)],
    email: "user" + i + "@example.com"
  });
}
```

---

2. Check the Number of Documents

db.users.countDocuments()

```
myDatabase> db.users.countDocuments()
10004
myDatabase>
```

Ensure it shows **10,000** records.

Step 1: Query Execution WITHOUT Index

Run this query **before indexing**:

db.users.find({ name: "User5000" }).explain("executionStats")

```
executionTimeMillis: 5,
totalKeysExamined: 0,
totalDocsExamined: 10004,
executionStages: {
  stage: 'COLLSCAN',
  filter: { name: { '$eq': 'User5000' } },
  nReturned: 1,
```

Check These Stats:

- **Total Documents Scanned:** Look at `"totalDocsExamined"` (should be high).
- **Execution Time:** `"executionTimeMillis"` (should be slow).

Step 2: Create an Index

Now, create an index on the `name` field:

db.users.createIndex({ name: 1 })

```
myDatabase> db.users.createIndex({ name: 1 })
name_1
myDatabase>
```

Step 3: Query Execution WITH Index

Run the same query again:

db.users.find({ name: "User5000" }).explain("executionStats")

Compare the Stats:

1. **Before Indexing:**

   - `"totalDocsExamined": 10000` (Scans all documents)
   - `"executionTimeMillis":5ms` (Higher value)
   - **Query uses COLLSCAN (Collection Scan)**

```
executionTimeMillis: 5,
totalKeysExamined: 0,
totalDocsExamined: 10004,
executionStages: {
  stage: 'COLLSCAN',
  filter: { name: { '$eq': 'User5000' } },
  nReturned: 1,
```

2. **After Indexing:**

   - `"totalDocsExamined": 1` (Scans only 1 document)
   - `"executionTimeMillis": 0ms` (Significantly lower)
   - **Query uses IXSCAN (Index Scan)**

```
executionTimeMillis: 27,
totalKeysExamined: 1,
totalDocsExamined: 1,
executionStages: {
  stage: 'FETCH',
  nReturned: 1,
  executionTimeMillisEstimate: 11,
  works: 2,
  advanced: 1,
  needTime: 0,
  needYield: 0,
  saveState: 1,
  restoreState: 1,
  isEOF: 1,
  docsExamined: 1,
  alreadyHasObj: 0,
  inputStage: {
    stage: 'IXSCAN',
    nReturned: 1,
```

In the first query, there will be high initial execution time i.e 27ms here, but we find that upon repeated queries, the executionTimeMillis is nearly 0ms.

```
executionTimeMillis: 0,
totalKeysExamined: 1,
totalDocsExamined: 1,
executionStages: {
  stage: 'FETCH',
  nReturned: 1,
  executionTimeMillisEstimate: 0,
  works: 2,
  advanced: 1,
  needTime: 0,
  needYield: 0,
  saveState: 0,
  restoreState: 0,
  isEOF: 1,
  docsExamined: 1,
  alreadyHasObj: 0,
  inputStage: {
    stage: 'IXSCAN',
    nReturned: 1,
    executionTimeMillisEstimate: 0,
    works: 2,
    advanced: 1,
    needTime: 0,
    needYield: 0,
    saveState: 0,
```