UNIT2

Topic1:Introduction to Mongo DB

MongoDB is a popular, open-source, NoSQL database management system that uses a document-oriented data model. Unlike traditional relational databases, which store data in tables and rows, MongoDB stores data in collections of documents (similar to JSON objects). This allows for more flexibility in the structure of the data and scalability for large datasets.

Here are some key points about MongoDB:

1. Document-Oriented

- Data is stored in **documents**, which are essentially key-value pairs. These documents are stored in a format called **BSON** (Binary JSON), which is similar to JSON but supports additional data types like dates and binary data.
- A document can contain nested data and arrays, making it a good fit for applications that require flexible, dynamic data schemas.

2. Collections

- Documents are organized into **collections**. A collection is akin to a table in a relational database, but without a fixed schema.
- Since there's no fixed schema, each document in a collection can have a different structure, allowing for flexibility in storing data.

3. Scalability

- MongoDB is designed with **horizontal scaling** in mind. This means it can easily distribute data across multiple servers (called **sharding**) to handle large amounts of data and high throughput.
- This makes MongoDB a good choice for applications that need to scale quickly and handle large amounts of data.

4. NoSQL Characteristics

- MongoDB is a **NoSQL** database, which stands for "Not Only SQL." It is designed to handle unstructured or semi-structured data, unlike traditional relational databases that use a structured query language (SQL) and rigid schemas.
- It is well-suited for modern applications that need to store large amounts of data without a strict schema or that need to handle data that changes frequently.

5. Indexing

- MongoDB supports powerful indexing features, allowing for fast query performance even with large datasets. You can create indexes on fields to speed up search operations, and MongoDB

6. Aggregation

- MongoDB provides a powerful aggregation framework that allows users to perform complex queries and transformations on data. The aggregation pipeline allows for operations such as filtering, grouping, and sorting data in a flexible manner.

7. Data Consistency

- MongoDB offers **strong consistency** by default. However, it also allows for **eventual consistency** configurations for use cases that can tolerate some delay in data synchronization across distributed systems.

8. Query Language

- MongoDB uses its own query language for interacting with the database. While it's not SQL, it has a rich set of operators and methods to query and manipulate data.
- Common operations like **find()**, **insert()**, **update()**, and **delete()** are used to interact with the database.

9. Replication

- MongoDB supports **replication**, meaning that data can be replicated across multiple servers to ensure high availability and data redundancy. This is done through a **replica set**, a group of MongoDB instances that maintain the same data set.

10. Use Cases

- MongoDB is particularly suited for use cases where data is unstructured or semi-structured, such as:
    - Content management systems
    - Real-time analytics
    - Internet of Things (IoT) applications
    - Mobile and web applications

Example of MongoDB Document:

Here's an example of what a MongoDB document might look like in BSON format (essentially JSON):

```
{
  "_id": ObjectId("5f9b4f6e9f1b2d0c4a9a2d7e"),
  "name": "John Doe",
```

```
  "email": "john.doe@example.com",
  "age": 29,
  "address": {
    "street": "123 Main St",
    "city": "Somewhere",
    "zipcode": "12345"
  },
  "interests": ["reading", "traveling", "sports"]
}
```

Topic2:Data Modelling in MongoDB

- **Data modeling in MongoDB** is the process of designing and creating the structure of collections and documents that will be stored in the database.

MongoDB Data modeling is the process of **arranging unstructured data** from a real-world event into a logical data model in a database.

There is no need to build a schema before adding data to **MongoDB database** because MongoDB is flexible. This means that MongoDB supports a **dynamic database schema**.

To create a perfect Data model in MongoDB, always balance **application needs**, **database engine performance features**, and **data retrieval patterns**.

When creating data models, always account for both the application uses of the data, such as queries, updates, and data processing, as well as the fundamental design of the data itself.

MongoDB Data Model Designs

For modeling data in MongoDB, two strategies are available. These strategies are different and it is recommended to analyze scenario for a better flow.

The two methods for data model design in MongoDB are:

1. Embedded Data Model
2. Normalized Data Model

1. Embedded Data Model

This method, also known as the **de-normalized** data model, allows you embedd all of the **related documents in a single document**.

These nested documents are also called **sub-documents.**

Embedded Data Model example

If we obtain student information in three different documents, Personal_details, Contact, and Address, we can embed all three in a single one, as shown below.

```
{
  _id: ,
  Std_ID: "987STD001"
  Personal_details:{
     First_Name: "Rashmika",
     Last_Name: "Sharma",
     Date_Of_Birth: "1999-08-26"
  },
  Contact: {
     e-mail: "rashmika_sharma.123@gmail.com",
```

```
    phone: "9987645673"
  },
  Address: {
    city: "Karnataka",
    Area: "BTM2ndStage",
    State: "Bengaluru"
  }
}
```

2. Normalized Data Model

In a normalized data model, object references are used to express the **relationships between documents and data objects**. Because this approach **reduces data duplication**, it is relatively simple to document **many-to-many relationships** without having to repeat content.

Normalized data models are the most effective technique to model large **hierarchical data** with cross-collection relationships.

Normalized Data Model Example

Here we have created multiple collections for storing students data which are linked with _id.

Student:

```
{
  _id: <StudentId101>,
  Std_ID: "10025AE336"
}
```

Personal_Details:

```
{
  _id: <StudentId102>,
  stdDocID: " StudentId101",
  First_Name: "Rashmika",
  Last_Name: "Sharma",
  Date_Of_Birth: "1999-08-26"
}
```

Contact:

```
{
  _id: <StudentId103>,
  stdDocID: " StudentId101",
  e-mail: "rashmika_sharma.123@gmail.com",
  phone: "9987645673"
}
```

Address:

```
{
  _id: <StudentId104>,
  stdDocID: " StudentId101",
  city: "Karnataka",
  Area: "BTM2ndStage",
  State: "Bengaluru"
}
```

Advantages Of Data Modeling in MongoDB

Data modeling in MongoDB is essential for a successful application, even though at first it might just seem like one more step. In addition to increasing **overall efficiency** and **improving development cycles**, data modeling helps you better understand the data at hand and identify future business requirements, which can save time and money.
In particular, applying suitable data models:

- Improves application performance through better database strategy, design, and implementation.
- Allows faster application development by making object mapping easier.
- Helps with better data learning, standards, and validation.
- Allows organizations to assess long-term solutions and model data while solving not just current projects but also future application requirements, including maintenance.4

Topic3:Advantages of MongoDB over RDBMS

MongoDB is a NoSQL database that offers several advantages over traditional Relational Database Management Systems (RDBMS). Some key benefits include:

1. **Schema-less Database:**
    - Unlike RDBMS, MongoDB does not require a predefined schema, allowing flexible and dynamic data structures.
2. **Scalability:**
    - MongoDB supports horizontal scaling (sharding), making it easier to handle large datasets and distribute data across multiple servers.
3. **High Performance:**
    - Due to its document-based storage, MongoDB offers faster read and write operations compared to RDBMS, especially for large-scale applications.
4. **Ease of Use & JSON-like Documents:**
    - Data is stored in BSON (Binary JSON), which is easier to work with in modern applications compared to rigid table structures in RDBMS.
5. **Better Handling of Large Volumes of Unstructured Data:**
    - Suitable for applications requiring frequent schema changes or dealing with diverse types of data.
6. **Indexing & Aggregation:**
    - MongoDB provides powerful indexing and aggregation framework, making queries more efficient compared to SQL-based databases.
7. **Replication & High Availability:**
    - MongoDB supports replication via replica sets, ensuring data redundancy and automatic failover.
8. **No Complex Joins:**
    - Unlike RDBMS, MongoDB avoids complex joins, improving performance by embedding related data within documents.

Topic4:Mongo Shell

- MongoDB Shell is a powerful, interactive command-line interface that enables developers and database administrators to perform operations and manage their MongoDB databases using JavaScript syntax. Whether you're a beginner just starting with MongoDB or an experienced user, MongoDB Shell offers a comprehensive environment for executing queries, testing commands, and handling database tasks.
- MongoDB Shell is a JavaScript-based tool that allows users to interact with MongoDB databases directly from the command line. It facilitates a wide range of operations from simple CRUD (Create, Read, Update, Delete) operations to more advanced database management tasks, making it an essential tool for anyone working with MongoDB.

MongoDB Shell allows developers to:

- Perform CRUD operations
- Query databases using rich syntax
- Execute aggregation queries for data analysis
- Handle large datasets efficiently
- Interact with the database in real-time using an interactive interface

MongoDB Shell Interface

The MongoDB Shell provides a direct interface between users and their MongoDB databases, making it easier to perform a wide variety of tasks. It simplifies operations like:

- Creating and deleting databases
- Inserting and modifying documents
- Running complex queries
- Managing collections

The interface uses JavaScript syntax, which allows developers to take advantage of its flexibility and execute complex queries. The immediate feedback and command history support make it perfect for both beginners and advanced users.

Key Features of MongoDB Shell

MongoDB Shell comes packed with essential features that enhance the database management experience. Here are some key highlights:

- Interactive Environment: MongoDB Shell offers an interactive experience where you can enter commands and see immediate results, making it ideal for learning and testing.
- JavaScript-Based: Since the shell uses JavaScript, you can leverage its rich capabilities to manipulate data and execute complex queries.
- Support for CRUD Operations: It perform all Create, Read, Update and Delete (CRUD) operations effortlessly through the shell.
- Aggregation Framework: Access advanced data processing features using the aggregation framework.
- Cross-Platform Compatibility: It use MongoDB Shell on various operating systems, including Windows, macOS and Linux.

Getting Started with MongoDB Shell

To begin using MongoDB Shell, simply install it on your system and launch it via the terminal or command prompt. Once connected, you can easily start interacting with your

MongoDB databases using various commands. To start using MongoDB Shell, follow these easy steps:

1. Install MongoDB Shell:
- Download the MongoDB Shell from the official MongoDB website.
- Follow the installation guide for your operating system (Windows, macOS, or Linux).

2. Launch the Shell
- Open your terminal or command prompt.
- Type mongosh to start the MongoDB Shell.

3. Access Help
- Type help within the shell to get a list of available commands and their descriptions.

Connecting to MongoDB using MongoDB Shell

To connect to a MongoDB deployment, you need a connection string. The connection string typically includes your MongoDB instance's address and authentication details.

For example, to connect to a MongoDB Atlas cluster, use the following command:

```
mongosh "mongodb+srv://mycluster.abcd1.mongodb.net/myFirstDatabase" --apiVersion 1
--username <username>
```

Once connected, you can start querying your database and performing various tas

Basic MongoDB Shell Commands

Let's explore some fundamental commands and operations you can perform using MongoDB Shell:

1. Show Databases: To view the list of databases, use the command:

```
show dbs
```

2. Switch Database: You can switch to a specific database using the use command:

```
use mydatabase
```

3. Show Collections: To list all collections within the current database, use:

```
show collections
```

4. Insert Document: To insert a document into a collection, use the insertOne() or insertMany() methods.

```
db.collection.insertOne({name: "Alice", age: 25})
```

5. Find Documents: To query documents in a collection:

```
db.collection.find()
db.collection.updateOne(
  { name: "Alice" },
  { $set: { age: 26 } }
)
```

Topic5: Configuration file in Mongo Shell

In MongoDB, the configuration file is typically used to define various settings and parameters that control how the MongoDB server operates. The file is usually in YAML format, and it allows you to specify options like network settings, storage configurations, security settings, and other server options.

Default Configuration File Location

- For **Linux**: `/etc/mongod.conf`
- For **Windows**: `C:\Program Files\MongoDB\Server\X.X\bin\mongod.cfg`

Basic Structure of `mongod.conf` File

Here's an example of a basic `mongod.conf` file:

```
# mongod.conf

# Where to store data.
storage:
  dbPath: /var/lib/mongodb
  journal:
    enabled: true

# how the MongoDB instance will listen for connections.
net:
  bindIp: 127.0.0.1  # Allows only local connections
  port: 27017

# Security settings
security:
  authorization: "enabled"  # Enables role-based access control

# Process management options
processManagement:
  fork: true  # Run as a daemon (in the background)

# Log settings
systemLog:
  destination: file
  path: /var/log/mongodb/mongod.log
  logAppend: true

# Replication settings (if used)
replication:
  replSetName: "rs0"  # Set for replica sets

# Sharding settings (if sharding is enabled)
sharding:
  clusterRole: "shardsvr"
```

```
# Set the path to the keyfile for sharding or replication security
security:
  keyFile: /etc/mongo-keyfile
```

Key Sections in the Configuration File

1. **Storage**
   - `dbPath`: The directory where MongoDB stores its data files.
   - `journal`: Enables or disables the write-ahead journal for data durability.
2. **Net (Network)**
   - `bindIp`: Specifies the IP address that MongoDB listens on. If set to `127.0.0.1`, MongoDB only accepts connections from the local machine.
   - `port`: Specifies the port MongoDB listens on (default is 27017).
3. **Security**
   - `authorization`: Enables or disables role-based access control (RBAC). If enabled, users must be authenticated to perform actions.
4. **Process Management**
   - `fork`: If set to true, MongoDB runs as a background process (daemon mode).
5. **System Log**
   - `destination`: Where MongoDB logs are stored (file or syslog).
   - `path`: File path to where logs are written.
   - `logAppend`: Appends logs to the existing log file.
6. **Replication** (for replica sets)
   - `replSetName`: The name of the replica set.
7. **Sharding** (for sharded clusters)
   - `clusterRole`: Can be set to "shardsvr" or "configsvr" depending on whether the node is part of a shard or a config server.

Topic 6: JSON File Format for storing documents:

In MongoDB, documents are stored in **BSON** format (Binary JSON), which is a binary representation of JSON. However, when you interact with MongoDB or export data, it is often represented as **JSON** for human readability.

Example of a JSON file format for storing MongoDB documents:

Here's a simple example of what a MongoDB document in a JSON file might look like:

```
[
  {
    "_id": {"$oid": "60b6e4d8f1b2c8e12e8b4567"},
    "name": "John Doe",
    "email": "john.doe@example.com",
    "age": 29,
    "address": {
      "street": "123 Main St",
      "city": "Anytown",
```

```
      "zipcode": "12345"
    },
    "interests": ["reading", "traveling", "sports"]
  },
  {
    "_id": {"$oid": "60b6e4d8f1b2c8e12e8b4568"},
    "name": "Jane Smith",
    "email": "jane.smith@example.com",
    "age": 34,
    "address": {
      "street": "456 Oak Rd",
      "city": "Othertown",
      "zipcode": "67890"
    },
    "interests": ["cooking", "music", "art"]
  }
]
```

In MongoDB, data is stored in **JSON-like** format called **BSON (Binary JSON)**. Here's how documents and collections work in MongoDB:

Topic7: Introduction to Documents and Collections

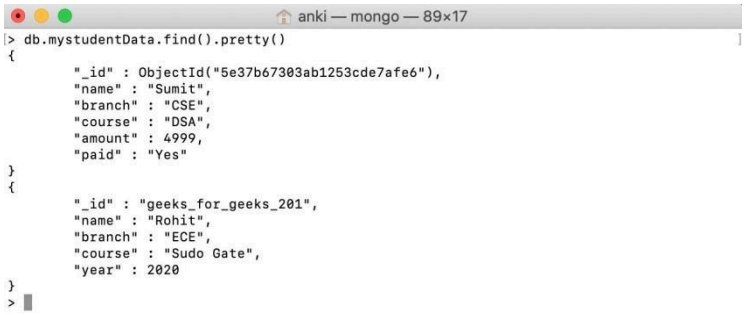
A Collection in MongoDB is similar to a table in **relational databases**. It holds a group of documents and is a part of a database. Collections provide structure to data, but like the rest of MongoDB, they are schema-less.

Schemaless

As we know that **MongoDB databases** are schemaless. So, it is not necessary in a collection that the schema of one document is similar to another document. Or in other words, a single collection contains different types of documents like as shown in the below example where **mystudentData** collection contain two different types of documents:

Multiple Collections per Database

A single database can contain multiple collections, each storing different types of documents.



Naming Restrictions for Collection:

Before creating a collection we should first learn about the naming restrictions for collections:

- Collection name must starts with an **underscore** (`_`) or a **letter** (a-z or A-Z)

- Collection name should not start with a number, and does not contain $, empty string, null character and does not begin with prefix `system.` as this is reserved for **MongoDB system collections**.
- The maximum length of the collection name is **120 bytes**(including the database name, dot separator, and the collection name).

Example:

```
db.books.insertOne({ title: "Learn MongoDB", author: "Jane Doe", year: 2023 })
```

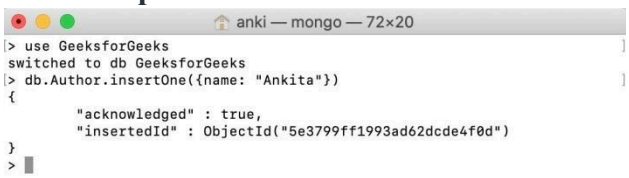
Creating collection

After creating database now we create a **collection** to store documents. The collection is created using the following syntax:

```
db.collection_name.insertOne({..})
```

Here, **insertOne()** function is used to store single data in the specified collection. And in the **curly braces {}** we store our data or in other words, it is a document.

For Example:



- In this example, we create a collection named as the **Author** and we insert data in it with the help of insertOne() function. Or in other words, {name: "Ankita"} is a document in the **Author collection**, and in this document, the name is the key or field and "**Ankita**" is the value of this key or field.
- After pressing enter we got a **message**(as shown in the above image) and this message tells us that the data enters successfully (i.e., "acknowledge": true) and also assigns us an automatically created id.
- It is the **special feature** provided by MongoDB that every document provided a **unique id** and generally, this id is created automatically, but you are allowed to create your own id (must be unique).

What is a Document in MongoDB?

In **MongoDB**, the data records are stored as BSON documents. Here, BSON stands for binary representation of **JSON documents**, although BSON contains more data types as compared to JSON. The document is created using **field-value pairs** or **key-value pairs** and the value of the field can be of any BSON type.

Syntax:

{
field1: value1
field2: value2
....
fieldN: valueN
}

Document Structure:

A document in MongoDB is a flexible data structure made up of **field-value pairs**. For instance:

```
{
 title: "MongoDB Basics",
 author: "John Doe",
 year: 2025
}
```

Naming restriction for Document Fields:

Before moving further first you should learn about the naming restrictions for fields:

- Fields in documents must be named with strings
- The _id field name is reserved to use as a primary key. And the value of this field must be unique, immutable, and can be of any type other than an array.
- The field name cannot contain null characters.
- The top-level field names should not start with a **dollar sign** ($).

Topic8:Database Commands in MongoDB

1. Show all databases:

```
show dbs
```

- **Explanation:**
  The `show dbs` command lists all the databases present in your MongoDB instance. It shows the names of all databases along with their size (storage size used).
  - This command does **not** show databases that are empty (i.e., databases without any collections).

Example Output:

```
admin  0.000GB
local  0.000GB
myDatabase  0.001GB
```

2. Create/Use a database:

```
use myDatabase
```

- **Explanation:**
  The `use <database_name>` command is used to **switch** to a specific database in MongoDB. If the database doesn't exist, MongoDB will **create** it the moment you insert the first document.
  - It's important to note that **no database is physically created** until you add data to it. Until then, the database remains empty, and MongoDB only considers it "selected."

Example:

- If you type `use myDatabase`, MongoDB will switch to the `myDatabase` database (if it exists) or create it when data is inserted.

3. Show all collections:

```
show collections
```

- **Explanation:**
  This command displays all **collections** within the current database. A collection is a grouping of MongoDB documents. It's similar to a table in a relational database system.

Example Output:

```
users
orders
products
```

Here, the collections `users`, `orders`, and `products` are shown as part of the current database.

4. Drop a database:

```
db.dropDatabase()
```

- **Explanation:**
  The `db.dropDatabase()` command is used to **delete** the currently selected database. This command **removes** the database along with all of its collections, documents, indexes, and data. It is irreversible—once dropped, you can't recover the database.

Example: If you are working in the `myDatabase` database and you run `db.dropDatabase()`, it will delete `myDatabase` and all of its collections.

Note:

- Make sure you really want to delete the database, as this action cannot be undone.

Topic9:Inserting and Saving Documents

In MongoDB, documents are the primary units of data storage. They are inserted or saved into collections using specific commands. Below is a **clear explanation** of the different ways to insert and save documents in MongoDB.

1. Inserting a Single Document:

The insertOne() method is used to insert **one document** into a collection.

Command:
db.<collection_name>.insertOne(<document>)

Explanation:

- The insertOne() method adds a **single document** to the specified collection.
- The document should be in **JSON-like format** (BSON format in MongoDB, but you typically use JSON-style syntax).
- MongoDB automatically generates an _id field for each document if you don’t specify one.

Example:

```
db.users.insertOne({
  "_id": 1,
  "name": "John Doe",
  "email": "john.doe@example.com"
})
```

- This inserts a single document into the users collection with the specified fields: _id, name, and email.
- **Note:** If you don't provide the _id, MongoDB will automatically generate it.

What Happens:

- MongoDB inserts the document, and the document will be stored in the users collection.

---

2. Inserting Multiple Documents:

The insertMany() method allows you to insert **multiple documents** at once into a collection.

Command:

```
db.<collection_name>.insertMany([<document1>, <document2>, ...])
```

Explanation:

- The insertMany() method inserts **multiple documents** into a collection at the same time.
- The documents should be provided as an array of JSON-like objects.
- MongoDB automatically generates _id values for each document if they are not specified.

Example:

```
db.users.insertMany([
  { "name": "Jane Doe", "email": "jane.doe@example.com" },
  { "name": "Alice Smith", "email": "alice.smith@example.com" }
])
```

- This command inserts two documents into the users collection: one for "Jane Doe" and one for "Alice Smith."

What Happens:

- MongoDB inserts both documents into the collection in a single operation.
- Each document will have a unique _id field generated by MongoDB.

---

3. Saving a Document:

The save() method is used to insert a document or update an existing document in the collection.

Command:
db.<collection_name>.save(<document>)

Explanation:

- The save() method behaves differently depending on whether the document already exists:
    - If the document **has an _id** and a document with that _id already exists in the collection, MongoDB will **update** the existing document.
    - If the document **doesn't have an _id**, MongoDB will **insert** it as a new document.

Example:

```
db.users.save({
  "_id": 1,
  "name": "John Doe",
  "email": "john.doe@example.com"
})
```

- In this case, MongoDB will **check if a document with _id: 1 already exists**.
    - If it does, MongoDB will **update** that document with the new fields (name and email).
    - If it doesn't, MongoDB will **insert** a new document with the specified _id.

What Happens:

- If a document with the same _id is found, it gets updated. Otherwise, a new document is inserted.

Note:

- The save() method is somewhat older and has been replaced by insertOne() and updateOne() methods for better clarity and control.

---

4. Inserting with a Custom _id:

In MongoDB, each document gets an automatically generated _id field by default. However, you can also provide a custom _id field if you want to manually control the unique identifier for each document.

Example:

```
db.users.insertOne({
  "_id": "user123",
  "name": "David Green",
  "email": "david.green@example.com"
})
```

- Here, a custom _id is provided ("user123") instead of MongoDB generating one.

What Happens:

- The document will be inserted into the collection with the custom _id value. If a document with the same _id already exists, it will result in an error (because _id must be unique within a collection).

Method	Description	Example
insertOne()	Inserts a single document into a collection.	db.users.insertOne({name: "John", email: "john@example.com"})
insertMany()	Inserts multiple documents into a collection in one operation.	db.users.insertMany([{name: "Jane", email: "jane@example.com"}, {...}])
save()	Inserts a document or updates it if a document with the same _id exists.	db.users.save({_id: 1, name: "John", email: "john@example.com"})
custom _id	Allows you to specify a custom _id for your document.	db.users.insertOne({_id: "user123", name: "David", email: "david@example.com"})

Topic10:Inserting Multiple documents

In MongoDB, you can insert multiple documents into a collection in a single operation using the `insertMany()` method. This method is useful when you have a batch of documents to insert at once, improving efficiency and performance over inserting documents one by one.

Inserting Multiple Documents in MongoDB

Command:

```
db.<collection_name>.insertMany([<document1>, <document2>, ...])
```

- **Explanation:**
  - `insertMany()` accepts an array of documents (objects).
  - Each document inside the array is inserted into the collection as a separate entry.
  - MongoDB automatically generates a unique `_id` for each document if you don't specify one.

Example:

Let's say you want to insert multiple user records into the `users` collection:

```
db.users.insertMany([
  { "name": "John Doe", "email": "john.doe@example.com", "age": 30 },
  { "name": "Alice Smith", "email": "alice.smith@example.com", "age": 25 },
  { "name": "David Green", "email": "david.green@example.com", "age": 35 }
])
```

In this example:

- The `insertMany()` method will insert **three documents** into the `users` collection.
- Each document contains fields for `name`, `email`, and `age`.
- MongoDB will automatically assign a unique `_id` to each document if you do not provide one.

What Happens:

- The three documents will be inserted into the collection in one operation.
- The `_id` fields will be automatically generated by MongoDB for each document.

Handling Errors in `insertMany()`

- If **any document** in the array fails to insert due to a duplicate key error or invalid format, MongoDB will throw an error and **abort the entire operation** by default.
- However, you can use the **`ordered`** option to control this behavior.

`ordered: true` (default behavior)

- MongoDB will stop inserting documents as soon as it encounters an error, and none of the documents after the error will be inserted.

```
db.users.insertMany([
  { "name": "John Doe", "email": "john.doe@example.com" },    // Will insert
  { "name": "Jane Doe", "email": "john.doe@example.com" },    // Will fail
(duplicate email)
  { "name": "Alice", "email": "alice@example.com" }            // Will not be
inserted
])
```

In this case, if the second document fails due to a duplicate key (`email`), MongoDB will **stop** and **not insert any other documents**.

`ordered: false` (continue on error)

- If you set `ordered: false`, MongoDB will **attempt to insert all documents**, even if one of them causes an error. Failed documents will be reported in the error message, but the rest will still be inserted.

```
db.users.insertMany([
  { "name": "John Doe", "email": "john.doe@example.com" },    // Will insert
  { "name": "Jane Doe", "email": "john.doe@example.com" },    // Will fail
(duplicate email)
  { "name": "Alice", "email": "alice@example.com" }            // Will insert
], { ordered: false })
```

- In this example, MongoDB will insert the first and third documents, but it will **skip** the second one due to the duplicate email and continue processing the rest.

Insert Result:

After you insert multiple documents using `insertMany()`, MongoDB will return an object with details about the insertion.

Example Output:

```
{
  "acknowledged" : true,
  "insertedIds" : {
    "0" : ObjectId("60a42bcdf4bcd8a3c8d74e28"),
    "1" : ObjectId("60a42bcdf4bcd8a3c8d74e29"),
    "2" : ObjectId("60a42bcdf4bcd8a3c8d74e2a")
  }
}
```

- The `acknowledged: true` means the operation was successful.
- The `insertedIds` field shows the unique `_id` values for each document that was inserted.

Example of Inserting Multiple Documents:

```
db.users.insertMany([
  { "name": "John Doe", "email": "john.doe@example.com", "age": 30 },
  { "name": "Alice Smith", "email": "alice.smith@example.com", "age": 25 },
  { "name": "David Green", "email": "david.green@example.com", "age": 35 },
  { "name": "Emily White", "email": "emily.white@example.com", "age": 28 }
])
```