UNIT - 4 NOTES

Unit 4: MongoDB

1. What is MongoDB?

MongoDB is an open-source NoSQL database written in C++ language. It uses JSON-like documents with optional schemas.

- It provides easy scalability and is a cross-platform, document-oriented database.
- MongoDB works on the concept of Collection (similar to tables) and Document (similar to rows/JSON objects).
- It combines the ability to scale out horizontally with features such as secondary indexes, range queries, sorting, aggregations, and geospatial indexes.
- MongoDB is developed by MongoDB Inc. and licensed under the Server Side Public License (SSPL).

2. What are the Key Features of MongoDB?

- **Indexing:** It supports generic secondary indexes and provides unique, compound, geospatial, and full-text indexing capabilities as well. This improves query performance significantly.
- **Aggregation:** It provides an aggregation framework based on the concept of data processing pipelines, allowing for complex data analysis and transformation.
- **Special collection and index types:** It supports time-to-live (TTL) collections for data that should expire automatically at a certain time.
- **File storage:** It supports an easy-to-use protocol (GridFS) for storing large files (like images, videos) and file metadata.
- **Sharding:** Sharding is the process of splitting data up across multiple machines (horizontal scaling) to handle large datasets and high throughput.

3. How to Perform Basic CRUD Operations in MongoDB?

CRUD stands for Create, Read, Update, Delete. Here's how to perform these operations using the MongoDB shell (`mongosh`):

- **a. How to Add/Insert Data?**
  - The basic method for adding data to MongoDB is "inserts".
  - **Insert a Single Document:** Use the collection's `insertOne` method:

```
> db.books.insertOne({"title" : "Start With Why"})
```

- **Insert Multiple Documents:** For inserting multiple documents into a collection, use `insertMany`. This method enables passing an array of documents to the database.

```
> db.users.insertMany([
...   { "name": "Alice", "age": 30 },
...   { "name": "Bob", "age": 28, "city": "New York" }
... ])
```

- **b. How to Update Data?**
  - Once a document is stored in the database, it can be changed using one of several update methods: `updateOne`, `updateMany`, and `replaceOne`.
  - `updateOne` and `updateMany` each take a filter document as their first parameter (to match documents) and a modifier document (using update operators like `$set` to describe changes) as the second parameter.

```
// Updates the age of the first user found with name "Alice"
> db.users.updateOne({ "name": "Alice" }, { $set: { "age": 31 } })

// Adds a status field to all users older than 25
> db.users.updateMany({ "age": { $gt: 25 } }, { $set: { "status": "active" }
})
```

  - `replaceOne` also takes a filter as the first parameter, but the second parameter is a *new document* that will completely replace the original document matching the filter (except for the immutable `_id` field).
  - For example, to replace a document like this:

```
{
  "_id" : ObjectId("4b2b9f67a1f631733d917a7a"), // This ID remains
  "name" : "alice",
  "friends" : 24,
  "enemies" : 2
}
```

    You could use `replaceOne` with a new document structure:

```
> db.users.replaceOne(
...   { "name": "alice" }, // Filter
...   { "username": "alice_new", "level": 5, "loggedIn": true } //
Replacement document
... )
```

- **c. How to Delete Data?**
  - The CRUD API in MongoDB provides `deleteOne` and `deleteMany` for this purpose.

- Both methods take a filter document as their first parameter. The filter specifies criteria to match documents for removal.
- **Delete One Document:** Removes the first document matching the filter.

```
// Example: Deletes the book document with _id equal to 3
> db.books.deleteOne({"_id" : 3})

// Example: Deletes the first user document where the name is "Bob"
> db.users.deleteOne({ "name": "Bob" })
```

- **Delete Multiple Documents:** Removes all documents matching the filter.

```
// Example: Deletes all users younger than 22
> db.users.deleteMany({ "age": { $lt: 22 } })
```

- **d. How to Perform Queries (Read Data)?**
  - The `find()` method is used to perform queries in MongoDB.
  - Querying returns a subset of documents in a collection, from none to the entire collection.
  - Which documents get returned is determined by the first argument to `find`, which is a filter document specifying the query criteria. An empty filter `{}` matches all documents.
  - **Example:** Find all users with age 24.

```
> db.users.find({"age" : 24})
```

  - **Example:** Find all documents in the `users` collection.

```
> db.users.find()
```

  - You can use query operators (like `$gt` for greater than, `$lt` for less than) for more complex queries.

```
// Example: Find users older than 25
> db.users.find({ "age": { $gt: 25 } })
```

4. SQL vs NoSQL: Key Differences

SQL and NoSQL are two types of databases used for different purposes.

- **1. What is SQL? (Relational Databases)**
  - SQL (Structured Query Language) databases store data in tables with predefined schemas (structure).
  - ✅ **Characteristics:**
    - Structured & organized (tables, rows, columns)
    - Uses SQL queries for data manipulation (SELECT, INSERT, UPDATE, DELETE, JOIN, etc.)
    - ACID-compliant (Atomicity, Consistency, Isolation, Durability) ensuring transaction reliability.

    - Good for structured data and enforcing complex relationships between data.
  - ✅ **Examples of SQL Databases:** MySQL, PostgreSQL, Microsoft SQL Server, Oracle Database.
  - 📌 **Example SQL Table (Users):**

ID	Name	Age	Email
1	John	28	john@mail.com
2	Sarah	24	sarah@mail.com

  - 👉 **Query to get users older than 25:**

```
SELECT * FROM Users WHERE Age > 25;
```

- **2. What is NoSQL? (Non-Relational Databases)**
  - NoSQL databases store data in flexible formats without rigid schemas. MongoDB is a popular *document* NoSQL database.
  - ✅ **Characteristics:**
    - Schema-less & flexible (no predefined structure required for documents in a collection).
    - Uses different data models (document, key-value, column-family, graph).
    - Scales horizontally (easy to distribute data across multiple servers).
    - Good for large-scale data, real-time applications, and unstructured/semi-structured data.
  - ✅ **Types of NoSQL Databases:**

Type	Description	Example
Document	Stores data as JSON-like documents	MongoDB, CouchDB
Key-Value	Simple key-value pairs	Redis, DynamoDB
Column-Family	Stores data in columns, not rows	Cassandra, HBase
Graph	Stores relationships in nodes/edges	Neo4j, ArangoDB

  - 📌 **Example NoSQL (MongoDB Document):**

```
{
    "_id": 1,
    "name": "John",
    "age": 28,
    "email": "john@mail.com",
    "interests": ["coding", "music"]
}
```

  - 👉 **Query to get users older than 25 in MongoDB:**

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

- **3. Key Differences: SQL vs NoSQL**

Feature	SQL (Relational)	NoSQL (Non-Relational)
Data Model	Tables (Rows & Columns)	Documents, Key-Value, Graph, Columnar
Schema	Fixed Schema (Structured)	Dynamic Schema (Flexible)
Query Language	SQL (SELECT, JOIN, etc.)	NoSQL queries (MongoDB uses JSON-based queries)
Scalability	Vertical Scaling (More CPU/RAM)	Horizontal Scaling (Distributed systems)
Transactions	ACID-compliant (strong consistency)	BASE (Eventual Consistency) - often tunable
Best For	Complex relationships, structured data	Big Data, Real-time applications, unstructured data

- **4. When to Use SQL vs NoSQL?**

Use Case	SQL	NoSQL
Banking & Financial Apps	✅ Yes (Requires ACID)	❌ No (Generally, due to ACID needs)
Real-Time Analytics	❌ No (Can be slow at scale)	✅ Yes
E-commerce & Inventory	✅ Yes	✅ Yes (Depends on structure needs)
Social Networks	❌ No (Graph relations complex)	✅ Yes (Especially Graph databases)
Content Management	❌ No (Less flexible for content)	✅ Yes (Flexible structure benefits)

- 📌 **Final Thoughts**
  - ✅ SQL is best for structured data, applications requiring strict consistency and transactions, and complex relational integrity.
  - ✅ NoSQL is best for unstructured or semi-structured data, applications needing high scalability, flexibility, and performance for large datasets.
  - ✅ Hybrid Approach: Some systems use both SQL & NoSQL together (e.g., user profiles in MongoDB, financial transactions in SQL).

5. How to Create and Manage MongoDB?

MongoDB stores data in a flexible JSON-like format (BSON). Here's a guide on creating and managing a MongoDB database.

- **1. Install & Setup MongoDB**
  - 💻 **Install MongoDB:**
    - **Windows:** Download the MSI installer from the MongoDB Official Site and follow the installation wizard.
    - **Mac (using Homebrew):**

      ```
      brew tap mongodb/brew
      brew install mongodb-community@6.0 # Check for latest stable version
      ```

    - **Linux (Ubuntu/Debian):** Follow the official documentation for package manager installation. Example:

      ```
      sudo apt update
      sudo apt install -y mongodb
      ```

  - **Start MongoDB Server:** The MongoDB server process is `mongod`. You need to specify a data directory (`--dbpath`).

    ```
    # Example: Start mongod, telling it where to store data
    # Make sure the directory exists and mongod has permissions to write to it
    mongod --dbpath /path/to/your/data/directory
    # Common default paths: /data/db (Linux/Mac), C:\data\db (Windows)
    ```

    👉 Default MongoDB runs on port `27017`.

- **2. Connect to MongoDB**
  - Open the MongoDB shell (`mongosh`) by running the command in your terminal:

    ```
    mongosh
    ```

  - You should see output similar to this, indicating a successful connection to the server running on `localhost:27017`:

    ```
    Current Mongosh Log ID: ...
    Connecting to:          mongodb://127.0.0.1:27017/?
    directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1.x.x
    Using MongoDB:          6.0.x
    Using Mongosh:          1.x.x
    ...
    test>
    ```

- **3. Create a Database**
  - To create a database or switch to an existing one, use the `use` command.

    ```
    use myDatabase
    ```

- 📌 If the database `myDatabase` does not exist, MongoDB creates it *implicitly* only when you first store data (like inserting a document or creating a collection) into that database. Simply using `use` doesn't create it on disk immediately.

- **4. Create and Manage Collections**
  - MongoDB stores documents in collections (analogous to tables in SQL).
  - **Create a Collection Explicitly:**

```
db.createCollection("users")
```

  - **OR Create Implicitly:** A collection is automatically created if it doesn't exist when you insert the first document into it.

```
db.products.insertOne({ name: "Laptop", price: 1200 }) // 'products' collection created if it didn't exist
```

- **5. Insert Data (Covered in detail in Q3a)**
  - **Insert One Document:**

```
db.users.insertOne({ name: "Alice", age: 25, email: "alice@mail.com" })
```

  - **Insert Multiple Documents:**

```
db.users.insertMany([
  { name: "Bob", age: 28 },
  { name: "Charlie", age: 22, status: "new" }
])
```

- **6. Read Data (Find Documents) (Covered in detail in Q3d)**
  - **Find All Documents:**

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

  - **Find Specific Document:**

```
db.users.find({ name: "Alice" })
```

  - **Find Using Operators:**

```
// Find users older than 25
db.users.find({ age: { $gt: 25 } })
```

- **7. Update Documents (Covered in detail in Q3b)**
  - **Update One Document:**

```
db.users.updateOne(
  { name: "Alice" },      // Filter: Find user named Alice
```

```
  { $set: { age: 26 } }   // Update: Set age to 26
)
```

- **Update Multiple Documents:**

```
db.users.updateMany(
  { age: { $lt: 30 } },        // Filter: Find users younger than 30
  { $set: { status: "young" } } // Update: Set status to "young"
)
```

- **8. Delete Documents (Covered in detail in Q3c)**
  - **Delete One Document:**

```
db.users.deleteOne({ name: "Alice" })
```

  - **Delete Multiple Documents:**

```
db.users.deleteMany({ age: { $lt: 25 } }) // Deletes users younger than 25
```

- **9. Manage Databases & Collections**
  - **List All Databases:**

```
show dbs
```

  - **List All Collections in Current Database:**

```
show collections
```

  - **Delete a Collection:**

```
db.users.drop() // Deletes the 'users' collection
```

  - **Delete a Database:**
    - First, switch to the database you want to delete using the `use` command.
    - Then, execute `db.dropDatabase()`. **Warning:** This action is irreversible.

```
use myDatabase      // Switch to the target database
db.dropDatabase()   // Delete the current database ('myDatabase')
```

- **10. Connect MongoDB to Node.js (Basic Example)**
  - 📌 **Install MongoDB Node.js driver:**

```
npm install mongodb
```

  - 📌 **Create `server.js` (Example):**

```
const { MongoClient } = require("mongodb");

// Connection URL
```

```
const url = "mongodb://localhost:27017"; // Or your MongoDB connection string
const client = new MongoClient(url);

// Database Name
const dbName = "myDatabase";

async function run() {
    try {
        // Connect the client to the server
        await client.connect();
        console.log("Connected successfully to server");

        const db = client.db(dbName);
        const usersCollection = db.collection("users");

        // Example: Insert a document
        await usersCollection.insertOne({ name: "David", age: 35 });
        console.log("Inserted a document into the users collection");

        // Example: Find documents
        const result = await usersCollection.find().toArray();
        console.log("Found documents =>", result);

    } catch (err) {
        console.error("An error occurred:", err);
    } finally {
        // Ensures that the client will close when you finish/error
        await client.close();
        console.log("Connection closed.");
    }
}

run().catch(console.error);
```

- ✅ **Run it with:**

```
node server.js
```

- 📌 **Summary Table: Common Mongo Shell Commands**

Task	Command
Start MongoDB	`mongod --dbpath /path/to/data`
Connect Shell	`mongosh`

Task	Command
Switch/Create DB	`use myDatabase`
Create Collection	`db.createCollection("users")` (or implicit on insert)
Insert One	`db.users.insertOne({ name: "John" })`
Insert Many	`db.users.insertMany([{...}, {...}])`
Find All	`db.users.find()`
Find Specific	`db.users.find({ name: "John" })`
Update One	`db.users.updateOne({ name: "John" }, { $set: { age: 31 } })`
Update Many	`db.users.updateMany({ age: {$lt: 30} }, { $set: { status: "active" } })`
Delete One	`db.users.deleteOne({ name: "John" })`
Delete Many	`db.users.deleteMany({ age: { $lt: 25 } })`
List Databases	`show dbs`
List Collections	`show collections`
Drop Collection	`db.users.drop()`
Drop Database	`use targetDb; db.dropDatabase()`

6. How to Migrate Data into MongoDB?

Migrating data into MongoDB can be done using various methods depending on the data source.

- 1️⃣ **Migration from JSON/CSV Files**
  - 📝 **Import JSON Data:** If you have data in a file where each line is a separate JSON document, or a single JSON array. Use the `mongoimport` command-line tool.

```
# For a file containing a JSON array of documents
mongoimport --db myDatabase --collection users --file users.json --jsonArray

# For a file where each line is a separate JSON document (newline-delimited JSON)
# mongoimport --db myDatabase --collection logs --file logs.json
```

✅ **Example `users.json` (for `--jsonArray`):**

```
[
  { "name": "Alice", "age": 25, "email": "alice@mail.com" },
  { "name": "Bob", "age": 28, "email": "bob@mail.com" }
]
```

- 📄 **Import CSV Data:** For CSV files, specify the type and use `--headerline` if the first row contains field names.

```
mongoimport --db myDatabase --collection users --type csv --file users.csv --headerline
```

✅ **Example `users.csv`:**

```
name,age,email
Alice,25,alice@mail.com
Bob,28,bob@mail.com
```

- 2️⃣ **Migration from SQL Databases (e.g., MySQL, PostgreSQL)**
  - Since SQL (relational) and NoSQL (document) have different structures, data often needs transformation.
  - 🔁 **Steps:**
    1. **Export** SQL data into a standard format (CSV or JSON are common).
    2. **Transform** the data (if needed) to fit your desired MongoDB document schema (e.g., embedding related data instead of using foreign keys). This can be done with scripts.
    3. **Import** the transformed data using `mongoimport` or a custom script (like Node.js or Python).
  - 📌 **Example: Export MySQL Data to CSV:**
    Run a query like this in MySQL (ensure the server process has file write permissions to the specified path):

```
SELECT *
FROM users
INTO OUTFILE '/path/on/server/to/users.csv' -- Path must be accessible by the MySQL server
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n';
```

Then, transfer the CSV file and import into MongoDB using:

```
mongoimport --db myDatabase --collection users --type csv --file /path/to/users.csv --headerline
```

  - 🔥 **Alternative: Using a Node.js Script for Migration:**
    Install necessary drivers:

```
npm install mysql mongodb
```

Create a migration script (`migrate.js`):

```
const mysql = require("mysql");
const { MongoClient } = require("mongodb");
```

```
// MySQL Connection Config
const sqlConnection = mysql.createConnection({
    host: "localhost",
    user: "root",
    password: "your_sql_password",
    database: "mySQLDatabaseName"
});

// MongoDB Connection Config
const mongoUrl = "mongodb://localhost:27017";
const mongoClient = new MongoClient(mongoUrl);
const mongoDbName = "myDatabase";
const mongoCollectionName = "users";

async function migrateData() {
    try {
        // Connect to MongoDB
        await mongoClient.connect();
        console.log("Connected to MongoDB");
        const mongoDb = mongoClient.db(mongoDbName);
        const usersCollection = mongoDb.collection(mongoCollectionName);

        // Connect to MySQL
        sqlConnection.connect(err => {
            if (err) {
                console.error("Error connecting to MySQL:", err);
                mongoClient.close();
                return;
            }
            console.log("Connected to MySQL");

            // Query MySQL
            sqlConnection.query("SELECT * FROM users", async (error, results)
=> {
                if (error) {
                    console.error("Error querying MySQL:", error);
                } else {
                    console.log(`Fetched ${results.length} records from
MySQL.`);
                    if (results.length > 0) {
                        // Optional transformation can happen here
                        const usersToInsert = results.map(user => ({
```

```
                            // Map SQL columns to MongoDB fields
                            name: user.name,
                            age: user.age,
                            email: user.email,
                            sql_id: user.id // Example: Preserve original ID
                        }));

                        // Insert into MongoDB
                        const insertResult = await
usersCollection.insertMany(usersToInsert);
                        console.log(`Successfully inserted
${insertResult.insertedCount} documents into MongoDB.`);
                    } else {
                        console.log("No records found in MySQL table to
migrate.");
                    }
                }

                // Close connections
                sqlConnection.end();
                await mongoClient.close();
                console.log("Connections closed.");
            });
        });

    } catch (mongoError) {
        console.error("Error connecting to or operating on MongoDB:",
mongoError);
        if (sqlConnection && sqlConnection.state !== 'disconnected') {
            sqlConnection.end(); // Ensure MySQL connection is closed on
Mongo error
        }
        if (mongoClient) {
            await mongoClient.close(); // Ensure Mongo client is closed
        }
    }
}

migrateData();
```

Run the script:

```
node migrate.js
```

✅ This script fetches data directly from MySQL and inserts it into MongoDB, allowing for transformation logic within the script.

- 3️⃣ **Migration from Firebase (Firestore) to MongoDB**
  - Use the Firebase Admin SDK to export data, then import into MongoDB.
  - 📌 **Install Dependencies:**

```
npm install firebase-admin mongodb
```

  - 📌 **Export/Import Script (`firebase-migrate.js`):**

```
const admin = require("firebase-admin");
const { MongoClient } = require("mongodb");

// --- Firebase Setup ---
// Download your service account key JSON file from Firebase Console
const serviceAccount = require("./path/to/your-firebase-
serviceAccountKey.json");

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
  // databaseURL: "https://your-project-id.firebaseio.com" // For Realtime
Database
});

const firestore = admin.firestore();
const firebaseCollectionName = "users"; // Collection in Firestore

// --- MongoDB Setup ---
const mongoUrl = "mongodb://localhost:27017";
const mongoClient = new MongoClient(mongoUrl);
const mongoDbName = "myDatabase";
const mongoCollectionName = "users"; // Collection in MongoDB

async function migrateFirebaseToMongo() {
    try {
        // Connect to MongoDB
        await mongoClient.connect();
        console.log("Connected to MongoDB");
        const db = mongoClient.db(mongoDbName);
        const mongoCollection = db.collection(mongoCollectionName);

        // Get data from Firestore
        const snapshot = await
```

```
firestore.collection(firebaseCollectionName).get();

        if (snapshot.empty) {
            console.log('No matching documents in Firestore.');
            return;
        }

        const firestoreDocs = snapshot.docs.map(doc => ({
            _id: doc.id, // Use Firestore doc ID as MongoDB _id
            ...doc.data() // Spread the document data
        }));

        console.log(`Fetched ${firestoreDocs.length} documents from
Firestore.`);

        // Insert into MongoDB
        if (firestoreDocs.length > 0) {
            const result = await mongoCollection.insertMany(firestoreDocs);
            console.log(`Successfully inserted ${result.insertedCount}
documents into MongoDB.`);
        }

    } catch (error) {
        console.error("Migration error:", error);
    } finally {
        await mongoClient.close();
        console.log("MongoDB connection closed.");
        // Firebase Admin SDK doesn't require explicit closing for this use
case
    }
}

migrateFirebaseToMongo();
```

Run the script:

```
node firebase-migrate.js
```

✅ This script extracts data from a Firestore collection and moves it into MongoDB.

- 4️⃣ **Migration from Excel to MongoDB**
  - 📌 **Steps:**
    1. Convert the Excel file (.xlsx) to CSV or JSON format using Excel itself or a script.
    2. Import the resulting CSV/JSON file into MongoDB using `mongoimport`.
    3. Alternatively, use a Node.js script with a library to read the Excel file directly.

- 🔥 **Using Node.js to Import Excel:**
  Install dependencies:

```
npm install xlsx mongodb
```

Create script (`excel-to-mongo.js`):

```
const xlsx = require("xlsx");
const { MongoClient } = require("mongodb");

// --- Excel File Setup ---
const excelFilePath = "users.xlsx"; // Path to your Excel file
const sheetName = "Sheet1"; // Name of the sheet containing data

// --- MongoDB Setup ---
const mongoUrl = "mongodb://localhost:27017";
const mongoClient = new MongoClient(mongoUrl);
const mongoDbName = "myDatabase";
const mongoCollectionName = "usersFromExcel";

async function migrateExcelToMongo() {
    try {
        // Read Excel File
        const workbook = xlsx.readFile(excelFilePath);
        const worksheet = workbook.Sheets[sheetName];
        if (!worksheet) {
            throw new Error(`Sheet "${sheetName}" not found in
${excelFilePath}`);
        }
        const jsonData = xlsx.utils.sheet_to_json(worksheet); // Converts
sheet rows to JSON objects
        console.log(`Read ${jsonData.length} records from Excel sheet
"${sheetName}".`);

        if (jsonData.length === 0) {
            console.log("No data found in Excel sheet to import.");
            return;
        }

        // Connect to MongoDB
        await mongoClient.connect();
        console.log("Connected to MongoDB");
        const db = mongoClient.db(mongoDbName);
        const collection = db.collection(mongoCollectionName);
```

```
        // Insert data into MongoDB
        const result = await collection.insertMany(jsonData);
        console.log(`Successfully inserted ${result.insertedCount} documents
from Excel into MongoDB.`);

    } catch (error) {
        console.error("Migration error:", error);
    } finally {
        await mongoClient.close();
        console.log("MongoDB connection closed.");
    }
}

migrateExcelToMongo();
```

Run the script:

```
node excel-to-mongo.js
```

✅ Reads data directly from an Excel sheet and imports it into a MongoDB collection.

- **Summary of Migration Methods**

Source	Method(s)	Notes
JSON	`mongoimport --jsonArray` or `mongoimport` (per line)	Easiest for standard JSON formats.
CSV	`mongoimport --type csv --headerline`	Simple for flat, tabular data.
MySQL/PostgreSQL	Export to CSV/JSON -> `mongoimport` **OR** Custom Script (Node.js, Python)	Script needed for transformation.
Firebase Firestore	Firebase Admin SDK + Custom Script (Node.js)	Requires service account key.
Excel (XLSX)	Convert to CSV/JSON -> `mongoimport` **OR** `xlsx` library + Script (Node.js)	Script offers more control.

- 🚀 **Key Takeaways for Migration**
  - ✅ `mongoimport` is the quickest method for simple JSON and CSV files.
  - ✅ For SQL to NoSQL migrations, carefully consider data transformation to leverage MongoDB's document model (embedding vs referencing).
  - ✅ Use Node.js (or Python) scripts for complex transformations, data validation during migration, or migrating from APIs/other databases.
  - ✅ Always validate the imported data in MongoDB to ensure correctness and completeness. Clean data before or during migration if necessary.

7. How to Use MongoDB with Node.js?

MongoDB is a very popular database choice for Node.js applications due to its JavaScript-friendly nature (queries and data use JSON/BSON). This guide covers common integration steps.

- 1️⃣ **Install MongoDB and Node.js**
  - 📌 **Install MongoDB:** (Ensure MongoDB server is installed and running)
    - **Windows:** Download MSI from MongoDB site.
    - **Mac (Homebrew):** `brew tap mongodb/brew; brew install mongodb-community`
    - **Linux (Ubuntu/Debian):** `sudo apt update; sudo apt install -y mongodb`
  - 📌 **Start MongoDB Server:**

    ```
    mongod --dbpath /path/to/your/data # Make sure it's running
    ```

  - 📌 **Install Node.js:** Download from nodejs.org or use a version manager like NVM.
- 2️⃣ **Install MongoDB Driver in Node.js Project**
  - Create your Node.js project directory and initialize npm:

    ```
    mkdir my-mongo-node-app
    cd my-mongo-node-app
    npm init -y
    ```

  - Install the official MongoDB driver:

    ```
    npm install mongodb
    ```

- 3️⃣ **Connect Node.js to MongoDB**
  - Create a file (e.g., `db.js`) to handle the connection logic:

    ```
    // db.js
    const { MongoClient } = require("mongodb");

    const url = "mongodb://localhost:27017"; // Your MongoDB connection string
    const dbName = "myDatabase"; // The database you want to use
    const client = new MongoClient(url);

    let dbConnection;

    module.exports = {
      connectToServer: async function () {
        try {
          await client.connect();
          console.log("✅ Successfully connected to MongoDB.");
    ```

```
      dbConnection = client.db(dbName);
    } catch (err) {
      console.error("❌ Failed to connect to MongoDB", err);
      process.exit(1); // Exit if connection fails
    }
  },

  getDb: function () {
    if (!dbConnection) {
        throw new Error("Call connectToServer first!");
    }
    return dbConnection;
  }
};
```

- In your main application file (e.g., `app.js` or `server.js`), call `connectToServer`.

- 4️⃣ **Perform CRUD Operations in Node.js**
  - Create a file (e.g., `app.js`) to demonstrate CRUD:

```
// app.js
const db = require("./db");

async function main() {
  try {
    // Connect to the database
    await db.connectToServer();
    const database = db.getDb();
    const usersCollection = database.collection("users");

    // --- CRUD Operations ---

    // CREATE (Insert One)
    console.log("Inserting document...");
    const insertResult = await usersCollection.insertOne({
      name: "Alice",
      age: 25,
      email: "alice@example.com",
      createdAt: new Date()
    });
    console.log("Inserted document ID:", insertResult.insertedId);

    // READ (Find Multiple)
    console.log("Reading documents...");
    const users = await usersCollection.find({ age: { $gte: 25 }
```

```
}).toArray(); // Find users 25 or older
    console.log("Found users:", users);

    // READ (Find One)
    console.log("Reading one document...");
    const specificUser = await usersCollection.findOne({ name: "Alice" });
    console.log("Found specific user:", specificUser);



    // UPDATE (Update One)
    console.log("Updating document...");
    const updateResult = await usersCollection.updateOne(
      { name: "Alice" },            // Filter
      { $set: { age: 26 } }         // Update operation
    );
    console.log("Matched:", updateResult.matchedCount, "Modified:",
updateResult.modifiedCount);

    // DELETE (Delete One)
    console.log("Deleting document...");
    const deleteResult = await usersCollection.deleteOne({ name: "Alice" });
    console.log("Deleted count:", deleteResult.deletedCount);

    console.log("CRUD Operations completed!");

  } catch (e) {
    console.error("An error occurred in main:", e);
  } finally {
    // The connection closing logic might be handled elsewhere,
    // e.g., on server shutdown if using Express.
    // For a simple script, client.close() would be here or in db.js.
    // For this example, assume connection managed by db.js or app lifecycle.
  }
}

main();
```

- Run the script:

```
node app.js
```

✅ This demonstrates basic C.R.U.D operations using the Node.js driver.

- 5️⃣ **MongoDB with Express.js API**
  - Build a simple REST API using Express to interact with MongoDB.

- 📌 **Install Dependencies:**

```
npm install express mongodb cors body-parser # cors & body-parser are common middleware
```

- 📌 **Create `server.js`:**

```
// server.js
const express = require("express");
const cors = require("cors"); // Cross-Origin Resource Sharing middleware
const bodyParser = require("body-parser"); // To parse request bodies
const db = require("./db"); // Reuse our db connection module

const app = express();
const port = 3000;

// Middleware
app.use(cors());
app.use(bodyParser.json()); // Parse JSON request bodies

let database; // Variable to hold the DB connection

// Start server only after DB connection is established
db.connectToServer()
  .then(() => {
    database = db.getDb(); // Get the database connection object

    // --- API Routes ---

    // GET All Users
    app.get("/users", async (req, res) => {
      try {
        const users = await database.collection("users").find().toArray();
        res.json(users);
      } catch (e) {
        res.status(500).send("Error fetching users: " + e.message);
      }
    });

    // GET User by Name (Example with URL parameter)
     app.get("/users/:name", async (req, res) => {
        try {
            const userName = req.params.name;
            const user = await database.collection("users").findOne({ name:
```

```
userName });
            if (user) {
                res.json(user);
            } else {
                res.status(404).send("User not found");
            }
        } catch (e) {
            res.status(500).send("Error fetching user: " + e.message);
        }
    });

    // POST (Create) User
    app.post("/users", async (req, res) => {
      try {
        const newUser = req.body; // Assumes JSON like { "name": "Bob",
"age": 30 }
        const result = await database.collection("users").insertOne(newUser);
        res.status(201).json({ message: "User added!", insertedId:
result.insertedId });
      } catch (e) {
         res.status(500).send("Error adding user: " + e.message);
      }
    });

    // PUT (Update) User by Name
    app.put("/users/:name", async (req, res) => {
      try {
        const userName = req.params.name;
        const updates = req.body; // Assumes JSON like { "age": 32 }
        const result = await database.collection("users").updateOne(
          { name: userName },
          { $set: updates }
        );
        if (result.matchedCount > 0) {
            res.json({ message: "User updated!", modifiedCount:
result.modifiedCount });
        } else {
            res.status(404).send("User not found for update");
        }
      } catch (e) {
        res.status(500).send("Error updating user: " + e.message);
      }
    });
```

```
    // DELETE User by Name
    app.delete("/users/:name", async (req, res) => {
      try {
        const userName = req.params.name;
        const result = await database.collection("users").deleteOne({ name:
userName });
         if (result.deletedCount > 0) {
            res.json({ message: "User deleted!", deletedCount:
result.deletedCount });
        } else {
            res.status(404).send("User not found for deletion");
        }
      } catch (e) {
        res.status(500).send("Error deleting user: " + e.message);
      }
    });

    // Start listening for requests
    app.listen(port, () => {
      console.log(`🚀 Server running on http://localhost:${port}`);
    });
  })
  .catch(err => {
    console.error("Failed to start server:", err);
  });
```

- **Run the API Server:**

```
node server.js
```

- ✅ **Test the API:** Use tools like Postman, `curl`, or your web browser:
  - `GET http://localhost:3000/users` -> Retrieve all users.
  - `POST http://localhost:3000/users` with JSON body `{ "name": "Bob", "age": 30 }` -> Add Bob.
  - `PUT http://localhost:3000/users/Bob` with JSON body `{ "age": 32 }` -> Update Bob's age.
  - `GET http://localhost:3000/users/Bob` -> Retrieve Bob's details.
  - `DELETE http://localhost:3000/users/Bob` -> Delete Bob.

- 🔥 **Summary: Node.js + MongoDB**

Task	Code/Command Example	Library/Tool
Install Driver	`npm install mongodb`	npm
Connect to DB	`const { MongoClient } = require("mongodb"); client.connect()`	`mongodb` driver
Select DB/Collection	`const db = client.db("dbName"); const col = db.collection("colName");`	`mongodb` driver
Insert Document	`collection.insertOne({ name: "Alice" })`	`mongodb` driver
Find Documents	`collection.find({ age: { $gt: 25 } }).toArray()`	`mongodb` driver
Update Document	`collection.updateOne({ name: "Alice" }, { $set: { age: 26 } })`	`mongodb` driver
Delete Document	`collection.deleteOne({ name: "Alice" })`	`mongodb` driver
Run Express API Server	`node server.js`	Node.js
Create API Routes	`app.get('/users', async (req, res) => { ... });`	Express.js

8. What are the Key MongoDB Services?

MongoDB offers a range of services beyond the core database to help developers build, deploy, scale, and manage applications more effectively.

- 1️⃣ **MongoDB Atlas (Cloud Database)**
  - ✅ Fully managed, global cloud database service. Takes care of infrastructure management.
  - ✅ Available on major cloud providers: AWS, Azure, and Google Cloud.
  - ✅ **Features:** Automatic scaling (storage and compute), automated backups & point-in-time recovery, multi-region/multi-cloud deployments, built-in security features (network isolation, encryption, role-based access), performance monitoring and alerting, serverless instances.
  - ✅ **Best for:** Most cloud-native applications, developers wanting to focus on building apps rather than managing DB infrastructure, serverless architectures.
- 2️⃣ **MongoDB Enterprise Advanced (Self-Managed)**
  - ✅ A commercial, self-hosted version of MongoDB designed for enterprise deployments (on-premises or private cloud).
  - ✅ **Includes:** All Community Edition features plus advanced security (LDAP/Kerberos authentication, encryption at rest), enterprise-grade monitoring (Ops Manager), advanced analytics integrations, compliance certifications, and commercial support from MongoDB Inc.
  - ✅ **Best for:** Large organizations with strict security/compliance requirements, those needing fine-grained control over their database environment, or requiring commercial support for self-

hosted instances.

- 3️⃣ **MongoDB Community Edition (Self-Managed)**
  - ✅ The free, open-source version of MongoDB.
  - ✅ **Includes:** The core MongoDB database features, including CRUD operations, indexing, aggregation, replication (for high availability), and sharding (for horizontal scaling).
  - ✅ Can be self-hosted on your own servers (physical, VMs, or private cloud).
  - ✅ **Ideal for:** Local development, small to medium-sized projects, learning MongoDB, or applications where the advanced features/support of Enterprise/Atlas are not required.
- 4️⃣ **MongoDB Realm (Mobile & Application Development Platform)**
  - ✅ A platform including a mobile database (Realm DB) for iOS & Android that works offline-first.
  - ✅ Automatically syncs data between mobile devices and MongoDB Atlas in the backend.
  - ✅ Provides backend application services like Functions (serverless logic), Triggers (database event handlers), and GraphQL/REST APIs directly from Atlas data.
  - ✅ **Best for:** Building mobile applications (iOS/Android/Web) that need offline data access and seamless cloud synchronization, rapidly developing backend logic connected to Atlas.
- 5️⃣ **MongoDB Charts (Data Visualization)**
  - ✅ A tool to create real-time visualizations and interactive dashboards directly from data stored in MongoDB Atlas.
  - ✅ Allows creating charts (bar, line, map, etc.) using a drag-and-drop interface without needing to write SQL or complex code.
  - ✅ Can be embedded into applications or used for internal business intelligence.
  - ✅ **Best for:** Quickly visualizing MongoDB data, business intelligence reporting, creating embedded analytics.
- 6️⃣ **MongoDB Compass (GUI Tool)**
  - ✅ A graphical user interface (GUI) for interacting with MongoDB databases (Community, Enterprise, or Atlas).
  - ✅ **Features:** Visualize and explore data in collections, visually build and run queries and aggregation pipelines, create/manage indexes, analyze query performance (explain plans), view server status, manage users/roles (with appropriate permissions).
  - ✅ **Best for:** Developers and Database Administrators (DBAs) who prefer a visual interface over the command-line shell (`mongosh`) for managing and exploring data.
- 7️⃣ **MongoDB Atlas Search (Integrated Full-Text Search)**
  - ✅ A managed full-text search engine built directly into MongoDB Atlas, powered by Apache Lucene.
  - ✅ Allows creating sophisticated search indexes on Atlas data to enable features like text relevance ranking, autocomplete, faceting, and multi-language support within applications.

- ✅ Eliminates the need to run and manage a separate search engine (like Elasticsearch).
- ✅ **Best for:** Applications requiring rich text search capabilities on data stored in Atlas.

- 8️⃣ **MongoDB Atlas Data Lake (Query Data in Place)**
  - ✅ Allows you to run MongoDB Query Language (MQL) queries directly against data stored in cloud object storage (AWS S3, Azure Blob Storage, GCP Cloud Storage).
  - ✅ Supports various data formats (JSON, BSON, CSV, TSV, Avro, ORC, Parquet).
  - ✅ Query data in place without needing to move or transform it into a MongoDB collection first. Can also combine queries across Atlas clusters and Data Lake storage.
  - ✅ **Best for:** Analyzing large volumes of data stored cost-effectively in object storage, unifying queries across operational databases (Atlas) and historical/archived data (S3 etc.).
- 9️⃣ **MongoDB Backup & Restore Solutions**
  - ✅ MongoDB Atlas provides fully managed continuous backups and point-in-time recovery.
  - ✅ MongoDB Enterprise (with Ops Manager/Cloud Manager) provides automated backup and restore capabilities for self-hosted deployments.
  - ✅ Community Edition relies on manual backups using tools like `mongodump` and `mongorestore`.
  - ✅ **Best for:** Ensuring data protection, disaster recovery, and point-in-time restoration capabilities. Managed solutions (Atlas) simplify this significantly.
- 🔷 **Summary Table: MongoDB Services**

MongoDB Service	Description	Primary Use Case / Best For	Management
MongoDB Atlas	Fully managed cloud database (DBaaS)	Cloud applications, Serverless, Ease of Use	MongoDB Managed
MongoDB Enterprise	Advanced self-hosted version w/ support	Large businesses, Compliance, Security Needs	Self-Managed
MongoDB Community	Free, open-source core database	Development, Small projects, Self-hosting	Self-Managed
MongoDB Realm	Mobile DB & backend sync platform	Mobile & Web Apps (Offline-first, Sync)	Managed (Atlas)
MongoDB Charts	No-code data visualization tool	Business Intelligence, Embedded Analytics	Managed (Atlas)
MongoDB Compass	GUI for database management	Developers & DBAs (Visual Interaction)	Tool (Any DB)
MongoDB Atlas Search	Integrated full-text search engine	Applications needing rich search features	Managed (Atlas)
MongoDB Atlas Data Lake	Query data in cloud object storage	Big Data analytics on S3/Azure Blob/GCS	Managed (Atlas)
Backup & Restore	Data protection features	Disaster Recovery, Data Safety	Managed/Self

- **Final Thoughts on MongoDB Services**
  MongoDB provides a comprehensive ecosystem beyond just the database core. Services like Atlas significantly reduce operational overhead, while tools like Compass and Charts enhance developer productivity and data insights. Choosing the right service depends on factors like deployment environment (cloud vs. self-hosted), scale, required features (security, mobile sync, search), and budget.
