MONGODB

DEVELOPER

NOTES

👨🏻‍💻 Name: Mr. Rupesh Lal

🌐 Website: www.itmodem.com

📞 WhatsApp: +91 7870886355

💼 Profession: Software Engineer

**📘Chapter 1. What is MongoDB?**

**✅ Introduction to MongoDB**

**MongoDB** is a popular **NoSQL database** that stores data in a **flexible, JSON-like format** called **BSON**. It’s designed for high performance, scalability, and ease of development.

**✅ NoSQL vs SQL**

| **Feature** | **SQL (Relational)** | **NoSQL (MongoDB)** |
| --- | --- | --- |
| Data structure | Tables (rows & columns) | Collections (documents) |
| Schema | Fixed schema | Dynamic schema |
| Joins | Supported | Not typically used |
| Language | SQL | Query language (JS-like) |
| Use case | Complex relationships, strict data | Fast dev, unstructured or nested data |

**✅ Key Concepts**

* **Database** → container for collections
* **Collection** → group of related documents (like a table)
* **Document** → individual data item (like a row), stored in BSON format
* **BSON** = Binary JSON
  + Supports extra data types like Date, ObjectId

**Example Document:**

{

"\_id": "60a7f9...",

"name": "John Doe",

"email": "john@example.com",

"age": 30,

"isMember": true

}

**✅ BSON Format**

* Stands for **Binary JSON**
* Similar to JSON but includes types like:
  + ObjectId
  + Date
  + Decimal128
* Used internally by MongoDB for storage efficiency

**✅ MongoDB Setup**

**1. 🖥️ Local Setup**

* Download from <https://www.mongodb.com/try/download/community>
* Install and run:
* mongod *# starts MongoDB server*
* mongo *# starts MongoDB shell*

**2. ☁️ MongoDB Atlas (Cloud)**

* Go to <https://www.mongodb.com/cloud/atlas>
* Steps:
  1. Sign up & create a free cluster
  2. Add your IP to whitelist
  3. Create a database & user
  4. Get connection URI (e.g.):
  5. mongodb+srv://username:password@cluster0.mongodb.net/mydb
  6. Use in Mongoose:
  7. mongoose.connect('mongodb+srv://...')

**✅ Summary**

| **Concept** | **Description** |
| --- | --- |
| MongoDB | NoSQL database using BSON format |
| Collection | Like a SQL table |
| Document | JSON-like record |
| BSON | Binary JSON with extra data types |
| Setup | Local or MongoDB Atlas (cloud) |

**📘 Chapter 2. MongoDB Shell & Compass**

**✅ MongoDB Interfaces**

| **Tool** | **Description** |
| --- | --- |
| **Mongo Shell** | CLI to interact with MongoDB databases |
| **MongoDB Compass** | GUI tool to visualize and manage databases |

**✅ MongoDB Shell – Basic CRUD Commands**

Assume we’re using a database called testdb and a collection called users.

use testdb // Switch or create database

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

db.users.find() // Read

db.users.updateOne({ name: "John" }, { $set: { age: 30 } }) // Update

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

**🔹 Create Documents**

db.users.insertOne({ name: "Alice", age: 22 });

db.users.insertMany([

{ name: "Bob", age: 28 },

{ name: "Charlie", age: 35 }

]);

**🔹 Read Documents (Querying)**

db.users.find(); *// Get all users*

db.users.find({ age: { $gt: 25 } }); *// age > 25*

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

**🔹 Update Documents**

db.users.updateOne(

{ name: "Bob" },

{ $set: { age: 29 } }

);

db.users.updateMany(

{ age: { $lt: 30 } },

{ $inc: { age: 1 } } *// Increase age by 1*

);

**🔹 Delete Documents**

db.users.deleteOne({ name: "Charlie" });

db.users.deleteMany({ age: { $gt: 30 } });

**✅ Filtering, Sorting & Projections**

**🔸 Filtering**

db.users.find({ age: { $gte: 25, $lte: 35 } });

db.users.find({ name: { $in: ["Alice", "Bob"] } });

**🔸 Sorting**

db.users.find().sort({ age: 1 }); *// Ascending*

db.users.find().sort({ name: -1 }); *// Descending*

**🔸 Projections**

db.users.find({}, { name: 1, \_id: 0 }); *// Show only name*

**✅ Aggregation Basics**

Aggregation is used for **data transformation and analysis**.

**🔸 Example: Group and Count by Age**

db.users.aggregate([

{ $group: { \_id: "$age", count: { $sum: 1 } } }

]);

**🔸 Example: Match + Project**

db.users.aggregate([

{ $match: { age: { $gte: 25 } } },

{ $project: { name: 1, age: 1, \_id: 0 } }

]);

**✅ MongoDB Compass (GUI)**

* Download from: <https://www.mongodb.com/products/compass>
* Use it to:
  + Visually browse collections
  + Run queries & aggregations
  + Inspect schemas
  + Export/import documents

**🧠 Summary**

| **Command** | **Function** |
| --- | --- |
| insertOne | Add a document |
| find | Query documents |
| updateOne | Update a document |
| deleteOne | Remove a document |
| sort | Sort results |
| project | Include/exclude fields |
| aggregate | Process/transform data |

**📘 Chapter 3. Mongoose ODM (Object Data Modeling)**

**✅ What is Mongoose?**

**Mongoose** is an **ODM library for Node.js** used to interact with MongoDB in an **object-oriented** way.

**🔹 Why Use Mongoose?**

| **Without Mongoose** | **With Mongoose** |
| --- | --- |
| Manual MongoDB syntax | Cleaner, object-style syntax |
| No schema enforcement | Schema-based data modeling |
| Weak validation | Built-in validation |

**✅ Installing Mongoose**

npm install mongoose

**✅ Connecting to MongoDB**

const mongoose = require('mongoose');

mongoose.connect('mongodb://127.0.0.1:27017/mydb', {

useNewUrlParser: true,

useUnifiedTopology: true

}).then(() => console.log("MongoDB connected"))

.catch(err => console.log(err));

**✅ Defining Schemas and Models**

**🔹 Schema**

Defines the structure of a document.

const mongoose = require('mongoose');

const { Schema } = mongoose;

const userSchema = new Schema({

name: { type: String, required: true },

age: Number,

email: { type: String, unique: true },

isAdmin: { type: Boolean, default: false }

});

**🔹 Model**

Compiles the schema into a usable class.

const User = mongoose.model('User', userSchema);

**✅ CRUD Operations with Mongoose**

**🔹 Create**

const user = new User({ name: "Alice", age: 24, email: "alice@mail.com" });

user.save();

or

User.create({ name: "Bob", age: 30, email: "bob@mail.com" });

**🔹 Read**

User.find(); *// All users*

User.findOne({ name: "Alice" });

User.findById("60a6f9..."); *// By \_id*

**🔹 Update**

User.updateOne({ name: "Bob" }, { age: 31 });

User.findByIdAndUpdate(id, { name: "Updated" }, { new: true });

**🔹 Delete**

User.deleteOne({ name: "Alice" });

User.findByIdAndDelete(id);

**✅ Validations**

Mongoose supports **built-in and custom validations**.

**🔹 Built-in**

const userSchema = new Schema({

name: { type: String, required: true, minlength: 3 },

age: { type: Number, min: 18 },

email: { type: String, match: /.+\@.+\..+/ }

});

**🔹 Custom Validation**

age: {

type: Number,

validate: {

validator: v => v % 2 === 0,

message: props => `${props.value} is not an even number!`

}

}

**✅ Middleware (Hooks)**

Mongoose lets you define **functions that run before or after** certain actions.

**🔹 Pre-save Hook**

userSchema.pre('save', function (next) {

console.log(`Saving user: ${this.name}`);

next();

});

**🔹 Post-save Hook**

userSchema.post('save', function (doc) {

console.log(`Saved: ${doc.name}`);

});

**🧠 Summary**

| **Concept** | **Description** |
| --- | --- |
| ODM | Object-based interaction with MongoDB |
| Schema | Structure definition |
| Model | Class for interacting with collection |
| Validation | Data rules and constraints |
| Middleware | Pre/post logic hooks for documents |

**📘Chapter 4. MongoDB Relationships with Mongoose**

**✅ 1. Why Relationships in MongoDB?**

MongoDB is **non-relational**, but Mongoose lets you **define relationships** between documents to model real-world data structures (e.g., users and posts, products and reviews).

There are two main strategies:

| **Strategy** | **Description** |
| --- | --- |
| Embedding | Nest related data inside a document |
| Referencing | Store ObjectId references to other docs |

**✅ 2. One-to-Many (Users → Posts)**

**🔸 Method 1: Embedding (Fast reads)**

const userSchema = new mongoose.Schema({

name: String,

posts: [

{

title: String,

content: String

}

]

});

Pros: Fewer queries Cons: Can grow large quickly, difficult to update nested docs

**🔸 Method 2: Referencing (Scalable)**

const postSchema = new mongoose.Schema({

title: String,

content: String,

author: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }

});

const userSchema = new mongoose.Schema({

name: String

});

Then save posts like:

const post = new Post({ title: 'Post 1', content: '...', author: user.\_id });

await post.save();

**✅ 3. populate() Method**

The .populate() method is used to **replace ObjectId with actual document data**.

Post.find()

.populate('author', 'name') *// Populate author field with name only*

.then(posts => console.log(posts));

Example output:

{

"title": "Post 1",

"author": {

"\_id": "60abc123...",

"name": "John Doe"

}

}

**✅ 4. Many-to-Many (Students ↔ Courses)**

const courseSchema = new mongoose.Schema({

name: String,

students: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Student' }]

});

const studentSchema = new mongoose.Schema({

name: String,

courses: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Course' }]

});

* Both documents store references to each other
* Use populate() to retrieve complete info on either side

Student.findOne().populate('courses').then(...);

Course.findOne().populate('students').then(...);

**🧠 Summary**

| **Relationship Type** | **Implementation Method** | **Example** |
| --- | --- | --- |
| One-to-many | Embed or Reference | User → Posts |
| Many-to-many | Reference both ways | Student ↔ Courses |
| Populate | .populate(fieldName) | Join-like behavior |

**📘Chapter 5. Indexes & Performance**

**✅ What is an Index?**

An **index** in MongoDB is similar to an index in a book — it helps MongoDB **find data faster** during queries.

Without an index, MongoDB **scans every document** in a collection (**collection scan**) → slow for large datasets.

**✅ Creating Indexes**

**🔹 In MongoDB Shell**

db.users.createIndex({ name: 1 }); *// Ascending*

db.users.createIndex({ age: -1 }); *// Descending*

**🔹 In Mongoose**

const userSchema = new mongoose.Schema({

name: { type: String, index: true }, *// index on name*

email: { type: String, unique: true } *// unique index*

});

**🔹 Compound Index (Multiple fields)**

db.orders.createIndex({ userId: 1, status: -1 });

**✅ Checking Indexes**

db.users.getIndexes();

**✅ Dropping Indexes**

db.users.dropIndex("name\_1"); *// Drop by index name*

**✅ Query Optimization Basics**

1. **Use indexes** on fields used in:
   * Filters (find({ name: 'John' }))
   * Sorts (sort({ age: 1 }))
   * Joins (populate('userId'))
2. **Use projection** to fetch only required fields:
3. db.users.find({}, { name: 1, \_id: 0 });
4. **Avoid $ne, regex, or non-indexed $or** → these don’t use indexes efficiently.
5. **Use .explain() to analyze queries**

db.users.find({ age: 30 }).explain("executionStats");

This tells you whether the query used an index and how many documents were scanned.

**✅ Index Types**

| **Type** | **Description** |
| --- | --- |
| Single Field | Basic index on one field |
| Compound Index | Index on multiple fields |
| Text Index | For full-text search on strings |
| Unique Index | Prevents duplicate values in the field |
| TTL Index | Auto-removes docs after a time (expires) |

**✅ Example: Text Index**

db.articles.createIndex({ title: "text", body: "text" });

db.articles.find({ $text: { $search: "mongodb" } });

**🧠 Summary**

| **Concept** | **Key Point** |
| --- | --- |
| Index | Makes queries faster |
| Mongoose Index | Define via schema or manually in MongoDB |
| optimize query | Use projection, indexes, and .explain() |
| index types | Single, compound, unique, text, TTL |

**📘 Chapter 6. Advanced Mongoose Concepts**

**✅ 1. Aggregation Pipeline in Mongoose**

The **aggregation pipeline** is used to process and transform data using stages like $match, $group, $project, etc.

**🔸 Example**

User.aggregate([

{ $match: { age: { $gte: 18 } } },

{ $group: { \_id: "$isAdmin", total: { $sum: 1 } } },

{ $sort: { total: -1 } }

]);

Use cases:

* Grouping & counting
* Calculating averages
* Filtering before/after transformations

**✅ 2. Virtuals**

**Virtuals** are fields **not stored in MongoDB**, but **computed** at runtime.

**🔸 Example: fullName virtual**

const userSchema = new mongoose.Schema({

firstName: String,

lastName: String

});

userSchema.virtual('fullName').get(function () {

return `${this.firstName} ${this.lastName}`;

});

Enable virtuals in output:

userSchema.set('toJSON', { virtuals: true });

**✅ 3. Discriminators**

Discriminators allow you to create **multiple models with different schemas** based on a common parent — useful for Single Collection Inheritance.

**🔸 Base + Subtype Models**

const options = { discriminatorKey: 'kind', collection: 'users' };

const BaseUser = mongoose.model('User', new mongoose.Schema({

name: String

}, options));

const Admin = BaseUser.discriminator('Admin', new mongoose.Schema({

accessLevel: Number

}));

const Customer = BaseUser.discriminator('Customer', new mongoose.Schema({

loyaltyPoints: Number

}));

Now both Admin and Customer will be saved in the same users collection.

**✅ 4. Transactions**

Mongoose supports **ACID transactions** when using MongoDB **replica sets**.

**🔸 Example: Transaction using session**

const session = await mongoose.startSession();

session.startTransaction();

try {

await User.create([{ name: 'Alice' }], { session });

await Order.create([{ item: 'Book', userId: someId }], { session });

await session.commitTransaction();

} catch (err) {

await session.abortTransaction();

} finally {

session.endSession();

}

Note: Transactions require MongoDB **replica set or Atlas cluster**

**🧠 Summary**

| **Feature** | **Purpose** |
| --- | --- |
| Aggregation | Transform/filter/group data |
| Virtuals | Computed fields not stored in DB |
| Discriminators | Subclassed models using a single collection |
| Transactions | ACID-compliant, multi-document operations |

**📘Chapter 7. Practice & Application**

**🔸 Exercises**

1. **Insert & UpdateData**
   * Insert multiple users with different roles and emails.
   * Add or update an isActive flag to all documents.
2. await User.insertMany([
3. { name: 'John', role: 'admin', email: 'john@mail.com' },
4. { name: 'Jane', role: 'editor', email: 'jane@mail.com' }
5. ]);
6. await User.updateMany({}, { $set: { isActive: true } });
7. **Write Queries**
   * Find all users with role admin
   * Find users who registered in the last 7 days
   * Count users grouped by role
8. const admins = await User.find({ role: 'admin' });
9. const recent = await User.find({
10. createdAt: { $gte: new Date(Date.now() - 7 \* 24 \* 60 \* 60 \* 1000) }
11. });
12. const grouped = await User.aggregate([
13. { $group: { \_id: "$role", count: { $sum: 1 } } }
14. ]);

**🔸 Mini Project: Blog DB with Comment System**

🧩 **Structure:**

* User
* BlogPost
* Comment

*// User Schema*

const User = mongoose.model("User", new Schema({

name: String,

email: String

}));

*// BlogPost Schema*

const BlogPost = mongoose.model("Post", new Schema({

title: String,

content: String,

author: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }

}));

*// Comment Schema*

const Comment = mongoose.model("Comment", new Schema({

content: String,

post: { type: mongoose.Schema.Types.ObjectId, ref: 'Post' },

author: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }

}));

✅ Features:

* Add users, posts, and comments
* Populate posts with authors and comments
* Query comments by post or user

const posts = await BlogPost.find().populate('author').populate({

path: 'comments',

populate: { path: 'author' }

});

**🔸 Interview Questions**

1. **How does .populate() work in Mongoose?**
   * It **replaces the ObjectId field** in a document with the full document from the referenced collection.
   * Example:
   * Post.find().populate('author', 'name email');
   * Behind the scenes, Mongoose performs an additional query using the referenced \_id.
2. **When should you use indexes in MongoDB?**
   * Use indexes when:
     + A field is **frequently used in find, sort, or join operations**
     + For **uniqueness enforcement** (e.g., emails)
     + To improve **query performance** in large collections
   * Avoid over-indexing—it slows down writes and consumes space.

**✅ Summary Table**

| **Practice Area** | **Task** |
| --- | --- |
| Exercises | Insert, update, filter, group |
| Mini Project | BlogPost + Comments + Users |
| Interview Focus | populate(), Index optimization |

**Top 50 Backend Interview Questions** on **Node.js**, **Express.js**, and **MongoDB**, covering basics to advanced topics:

**🔵 Node.js (18 Questions)**

1. What is Node.js? Why is it popular for backend development?
2. Explain the Node.js architecture. What is the Event Loop?
3. What is the difference between Node.js and traditional server-side platforms like PHP?
4. What are global objects in Node.js?
5. How does the require system work in Node.js?
6. What is the difference between CommonJS and ES Modules in Node?
7. How do you handle asynchronous operations in Node.js?
8. Explain callbacks, promises, and async/await.
9. What are Streams and Buffers in Node.js?
10. How do you handle file operations in Node using the fs module?
11. What is the process object in Node.js?
12. What is the role of package.json and package-lock.json?
13. What are devDependencies vs dependencies in npm?
14. How do you debug Node.js applications?
15. What is middleware in Node.js?
16. What are the different ways to create a server in Node.js?
17. What are environment variables, and how do you use them?
18. What is event-driven programming in Node.js?

**🟠 Express.js (16 Questions)**

1. What is Express.js? How does it work with Node.js?
2. How do you set up a basic Express server?
3. What are routes in Express? How are they defined?
4. How do you handle dynamic route parameters in Express?
5. What is the use of req.query vs req.params?
6. What is middleware in Express? Difference between built-in, third-party, and custom middleware?
7. What is the role of next() in middleware?
8. How do you serve static files in Express?
9. How do you structure an Express app using the MVC pattern?
10. What are Express routers and how do you modularize routes?
11. How do you handle errors and send custom error messages in Express?
12. How do you secure an Express app (helmet, rate-limiting, etc.)?
13. What is CORS and how do you handle it in Express?
14. How do you build a RESTful API using Express?
15. What tools can be used to test Express APIs (e.g., Postman, Jest)?
16. How do you use sessions or JWT for authentication in Express?

**🟢 MongoDB + Mongoose (16 Questions)**

1. What is MongoDB and how is it different from SQL databases?
2. What are documents and collections in MongoDB?
3. What is BSON?
4. How do you insert, update, and delete data in MongoDB?
5. What are the common query operators ($in, $gte, $or, etc.)?
6. How do you filter, sort, and project data in MongoDB?
7. What is Mongoose and why do we use it?
8. What is a Mongoose schema and model?
9. How do you perform CRUD operations using Mongoose?
10. What is the use of Mongoose virtuals?
11. How does the .populate() method work in Mongoose?
12. What are indexes in MongoDB? When should you use them?
13. What is the aggregation pipeline? Use cases?
14. What are Mongoose middleware/hooks?
15. How do you implement one-to-many or many-to-many relationships in MongoDB?
16. How do transactions work in MongoDB, and how are they used in Mongoose?

**30 Top Backend Exercise Questions** focused on **Node.js, Express.js, and MongoDB** — ideal for practice, interviews, and project readiness:

**🔵 Node.js (10 Exercise Questions)**

1. Create a Node.js script that reads a file and displays its content in the console.
2. Build a CLI tool in Node.js that takes input from the terminal and saves it to a file.
3. Create a module that exports a function to add two numbers and another function to subtract.
4. Set up a timer using setInterval and clearInterval.
5. Write a program that reads environment variables using process.env.
6. Create a logging module with levels like info, warn, and error.
7. Build a basic HTTP server that returns "Hello, World!".
8. Use fs to append data to a file every time the server is hit.
9. Simulate async file read using both Promise and async/await.
10. Create a stream that reads a large file and writes it to another file.

**🟠 Express.js (10 Exercise Questions)**

1. Set up a basic Express.js server with a / route returning JSON {message: "Welcome"}.
2. Create a RESTful API with routes: GET /users, POST /users, PUT /users/:id, DELETE /users/:id.
3. Handle route parameters and query strings (/search?name=John).
4. Implement middleware that logs the request method and path.
5. Create custom error-handling middleware and return JSON error messages.
6. Build an Express app that serves static HTML/CSS files.
7. Use express.json() to parse incoming JSON and respond with the data.
8. Modularize your routes using Express.Router().
9. Create authentication middleware that checks if a user token exists in headers.
10. Test your Express routes using Postman or Swagger.

**🟢 MongoDB + Mongoose (10 Exercise Questions)**

1. Connect to MongoDB Atlas using Mongoose.
2. Create a User model with fields: name, email, password.
3. Insert multiple users into the database using insertMany.
4. Create a GET route to fetch all users from MongoDB.
5. Write a query to fetch users who have registered within the last 30 days.
6. Update a user’s name by their \_id.
7. Delete a user using their email.
8. Implement .populate() to show posts with their author's details.
9. Create a blog post schema with embedded comments.
10. Use the aggregation pipeline to count users grouped by roles.