BACKEND

DEVELOPER

NOTES

👨🏻‍💻 Name: Mr. Rupesh Lal

🌐 Website: www.itmodem.com

📞 WhatsApp: +91 7870886355

💼 Profession: Software Engineer

**🧠 MASTER NOTES OUTLINE – Node.js, Express.js, MongoDB**

**Structure for each topic**:

✅ *Theory (Concepts)*

✅ *Syntax & Code Examples*

✅ *Exercises*

✅ *Mini Projects*

✅ *Interview Questions*

**🔵 SECTION 1: Node.js – Basics to Advanced**

**1. Introduction to Node.js**

* What is Node.js? Why use it?
* Node vs Browser
* Node.js Architecture
* Installing Node.js & npm

**2. Node.js Fundamentals**

* Modules (Built-in, Custom, Third-party)
* CommonJS vs ES Modules
* require, module.exports, import, export
* File system module (fs)
* Path module
* Events module
* OS module
* process object
* Streams and Buffers
* CLI tools

**3. Asynchronous Programming in Node**

* Callback functions
* Promises
* async/await
* Event Loop & Call Stack
* Error handling

**4. HTTP & Server Basics**

* Creating a basic HTTP server
* Handling requests/responses
* Status codes
* Serving HTML/JSON

**5. npm & Package Management**

* package.json & package-lock.json
* Installing packages
* Dev dependencies vs Prod dependencies
* Versioning (^, ~)

**6. Debugging & Logging**

* console.log, debug, node inspect, VSCode debugger
* Logging best practices

**🔸 Exercises: Create a file reader, HTTP server**

**🔸 Mini Project: Simple File Upload Server**

**🔸 Interview Qs: What is event-driven programming in Node?**

**🟠 SECTION 2: Express.js – Basics to Advanced**

**1. What is Express?**

* Why use Express with Node?
* Setting up a project
* Basic Express server

**2. Express Routing**

* app.get, app.post, app.put, app.delete
* Route parameters & query strings
* Express Router

**3. Middleware**

* Built-in vs Custom middleware
* express.json(), express.urlencoded()
* Error-handling middleware
* next() usage

**4. Templates & Static Files**

* Serving static files
* View engines (e.g., EJS)

**5. REST API with Express**

* Building CRUD APIs
* Status codes
* Validation and error messages
* Postman testing

**6. Express + MongoDB Integration**

* Connecting MongoDB with Mongoose
* Routes + DB CRUD Operations
* MVC Pattern

**7. Authentication (Intro)**

* Sessions & Cookies
* JWT Basics

**🔸 Exercises: Simple routing, custom middleware**

**🔸 Mini Project: To-Do App API**

**🔸 Interview Qs: Difference between middleware and route handler?**

**🟢 SECTION 3: MongoDB + Mongoose – Basics to Advanced**

**1. What is MongoDB?**

* NoSQL vs SQL
* Collections and Documents
* BSON format
* Setup (local & cloud - MongoDB Atlas)

**2. MongoDB Shell & Compass**

* Basic commands: insert, find, update, delete
* Filtering, sorting, projections
* Aggregation basics

**3. Mongoose ODM**

* What is Mongoose? Why use it?
* Schemas and Models
* CRUD operations with Mongoose
* Validations
* Middleware (Hooks)

**4. Relationships**

* One-to-many, many-to-many
* populate() method

**5. Indexes & Performance**

* Creating indexes
* Query optimization basics

**6. Advanced Mongoose**

* Aggregation pipeline
* Virtuals
* Discriminators
* Transactions

**🔸 Exercises: Insert & update data, write queries**

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

**🔸 Interview Qs: How does populate() work? When to use indexes?**

**🧩 FINAL SECTION: Mini Projects (Full-Stack Ready)**

1. 📝 Notes App (CRUD + MongoDB)
2. 📚 Bookstore API with Authentication
3. 📬 Contact Manager with Validation
4. 🛒 Simple E-commerce Backend
5. 🔐 Auth System using JWT + MongoDB

**❓ Interview Preparation (Bonus)**

* Top 30 Node.js Interview Questions
* Top 30 Express.js Interview Questions
* Top 30 MongoDB Questions
* Common Backend Scenarios (e.g., rate-limiting, error handling, auth flows)
* MCQs for Practice

**✅ BONUS TIPS**

* Document all **code snippets with comments**
* Use **diagrams** for architecture and flowcharts
* Keep a **Glossary** of common terms
* Add **“Common Mistakes” section** for each topic

**🔵 SECTION 1: Node.js – Basics to Advanced**

**📘 1. Introduction to Node.js**

**✅ What is Node.js?**

* **Node.js** is an **open-source**, **cross-platform** JavaScript runtime environment.
* It allows you to **run JavaScript code outside the browser**, primarily on the **server-side**.
* Built on **Google Chrome’s V8 JavaScript engine**.
* Created by **Ryan Dahl** in **2009**.

**Think of it like this**: Node.js lets you use JavaScript to write backend (server-side) code just like you use it on the frontend (browser).

**✅ Why Use Node.js?**

| **Feature** | **Description** |
| --- | --- |
| 🔁 Non-blocking I/O | Node uses **asynchronous** event-driven architecture. No waiting for one task to complete. |
| 🚀 Fast Performance | Built on V8 engine which compiles JavaScript to native machine code. |
| 🔗 Single Language | Use **JavaScript for both frontend and backend**, reducing learning curve. |
| 📦 Rich Ecosystem | Access to thousands of open-source libraries via **npm**. |
| ⚙️ Scalable | Good for building **real-time applications** (like chat apps, gaming, API services). |

🧠 Example: Apps like **Netflix, LinkedIn, PayPal** use Node.js for performance and scalability.

**✅ Node.js vs Browser JavaScript**

| **Feature** | **Browser** | **Node.js** |
| --- | --- | --- |
| Environment | Runs in browser | Runs on server |
| Global Object | window | global |
| APIs Available | DOM, alert, document | File system, HTTP, OS, Streams |
| Use Case | UI Interactions | Server, API, File handling |

**✅ Node.js Architecture**

Node.js uses a **Single-threaded Event Loop** architecture.

**Core Components:**

1. **Event Loop** – Manages async tasks using events and callbacks.
2. **V8 Engine** – Executes JavaScript code.
3. **libuv** – Provides async I/O, thread pool, and event loop.
4. **C++ Bindings** – Bridges low-level system access.
5. **Node APIs** – Expose functionalities like fs, http, os.

**Flow Example**: Client sends request → Node handles it using Event Loop → Uses callback/Promise → Sends response (non-blocking)

🖼️ *(Add diagram if needed: Event Loop → Callback Queue → Call Stack → Response)*

**✅ Installing Node.js and npm**

1. **Download Node.js**
   * Visit: [https://nodejs.org](https://nodejs.org/)
   * Choose **LTS version** (recommended for most users)
2. **Install it (Windows/macOS/Linux)**
3. **Verify Installation**

node -v *# shows Node.js version*

npm -v *# shows npm version*

1. **Basic Commands**

node *# enters Node REPL (interactive shell)*

node app.js *# runs JavaScript file*

npm init *# initializes package.json*

npm install <package> *# installs a package*

**📝 Summary**

* Node.js lets you build server-side apps using JavaScript.
* It’s fast, scalable, and non-blocking.
* Ideal for real-time, I/O-heavy applications.
* Installs with npm (Node Package Manager) for managing packages.

**🔵 SECTION 1: Node.js – Basics to Advanced**

**📘 2. Node.js Fundamentals**

**✅ 1. Modules in Node.js**

Node.js follows a **modular architecture**. Code is split into reusable parts called **modules**.

**a. 🔹 Built-in Modules**

Node provides core modules like:

* fs – File system operations
* path – Work with file and directory paths
* http – Create HTTP server
* events – Event-driven programming
* os – System information

**Example**:

const fs = require('fs');

**b. 🔹 Custom Modules**

Any .js file is a module.

*// greet.js*

function greet(name) {

return `Hello, ${name}`;

}

module.exports = greet;

*// app.js*

const greet = require('./greet');

console.log(greet('John'));

**c. 🔹 Third-party Modules**

Installed via **npm**.

npm install lodash

const \_ = require('lodash');

**✅ 2. CommonJS vs ES Modules**

| **Feature** | **CommonJS (require)** | **ES Modules (import)** |
| --- | --- | --- |
| File Extension | .js | .mjs or set "type": "module" in package.json |
| Import Syntax | const fs = require('fs') | import fs from 'fs' |
| Export Syntax | module.exports = value | export default value or export {} |
| Execution | Synchronous | Asynchronous |

Node uses **CommonJS by default**.

**✅ 3. require, module.exports, import, export**

**🔹 CommonJS:**

*// math.js*

function add(a, b) {

return a + b;

}

module.exports = add;

*// app.js*

const add = require('./math');

console.log(add(2, 3));

**🔹 ES Modules:**

*// math.mjs*

export function add(a, b) {

return a + b;

}

*// app.mjs*

import { add } from './math.mjs';

console.log(add(2, 3));

**✅ 4. File System Module (fs)**

Use fs to read/write files.

**Synchronous:**

const data = fs.readFileSync('file.txt', 'utf8');

**Asynchronous:**

fs.readFile('file.txt', 'utf8', (err, data) => {

if (err) throw err;

console.log(data);

});

**✅ 5. Path Module**

Helps to work with file paths.

const path = require('path');

console.log(path.basename(\_\_filename)); *// filename*

console.log(path.dirname(\_\_filename)); *// directory*

console.log(path.join(\_\_dirname, 'folder', 'file.txt'));

**✅ 6. Events Module**

Node is **event-driven**. Use EventEmitter to create and listen to events.

const EventEmitter = require('events');

const emitter = new EventEmitter();

emitter.on('greet', (name) => {

console.log(`Hello, ${name}`);

});

emitter.emit('greet', 'Alice');

**✅ 7. OS Module**

Provides system-level info.

const os = require('os');

console.log(os.platform()); *// e.g., 'win32'*

console.log(os.cpus()); *// CPU info*

console.log(os.totalmem()); *// Total memory*

**✅ 8. process Object**

Gives access to system environment and runtime.

console.log(process.env); *// Environment variables*

console.log(process.argv); *// CLI arguments*

console.log(process.cwd()); *// Current directory*

Example with CLI:

node app.js hello

console.log(process.argv[2]); *// 'hello'*

**✅ 9. Streams and Buffers**

Used for large data, e.g., file/video streams.

const fs = require('fs');

const readStream = fs.createReadStream('file.txt', 'utf8');

readStream.on('data', chunk => {

console.log(chunk);

});

* **Stream**: Handle data piece-by-piece.
* **Buffer**: Temporary memory for binary data.

**✅ 10. CLI Tools in Node.js**

Build command-line apps using process.argv, fs, etc.

*// cli.js*

const name = process.argv[2];

console.log(`Hello, ${name}`);

node cli.js Alice

*# Output: Hello, Alice*

**🧠 Summary**

* Node modules help organize and reuse code.
* Built-in modules are powerful: fs, path, events, os, etc.
* CommonJS (require) is default; ES Modules (import) are modern.
* Use streams for performance and buffers for binary data.
* You can build command-line tools using core Node.js features.

**🔵 SECTION 1: Node.js – Basics to Advanced**

**📘 3. Asynchronous Programming in Node.js**

**✅ What is Asynchronous Programming?**

In Node.js, asynchronous programming is used to handle tasks **without blocking the main thread**. It allows your app to **handle multiple operations simultaneously**, like reading a file, querying a database, or calling an API—all **without waiting for each task to finish**.

**✅ 1. Callback Functions**

A **callback** is a function passed as an argument to another function, to be executed later (after async work is done).

**Example:**

const fs = require('fs');

fs.readFile('file.txt', 'utf8', (err, data) => {

if (err) return console.error(err);

console.log(data);

});

**❗ Callback Hell**

When callbacks are nested within callbacks, it becomes hard to read and maintain:

doSomething(() => {

doSomethingElse(() => {

yetAnotherThing(() => {

*// 😵 deeply nested!*

});

});

});

**✅ 2. Promises**

A **Promise** is an object representing the **future completion (or failure)** of an asynchronous operation.

**States:**

* pending
* fulfilled
* rejected

**Basic Usage:**

const fetchData = () => {

return new Promise((resolve, reject) => {

setTimeout(() => resolve("Data received"), 1000);

});

};

fetchData()

.then(data => console.log(data))

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

**✅ 3. async / await**

Introduced in ES2017, async/await allows writing **async code that looks like synchronous code**. It’s built on top of Promises.

**Example:**

const fetchData = () => {

return new Promise((resolve) => {

setTimeout(() => resolve("Data received"), 1000);

});

};

async function main() {

try {

const result = await fetchData();

console.log(result);

} catch (err) {

console.error(err);

}

}

main();

✅ await can only be used inside async functions.

**✅ 4. Event Loop & Call Stack**

The **event loop** is what allows Node.js to perform non-blocking I/O operations.

**How it works:**

1. JavaScript runs in a **single thread**.
2. The **Call Stack** handles execution.
3. Async tasks (e.g. file read, API calls) go to the **Web APIs / Node APIs**.
4. When complete, callbacks are queued in the **Callback Queue**.
5. The **Event Loop** checks if the Call Stack is empty, then pushes queued callbacks into it.

🖼️ *(Use a diagram: Call Stack ⇄ Event Loop ⇄ Callback Queue)*

**✅ 5. Error Handling in Async Code**

**a. Callbacks:**

Always check err first:

fs.readFile('file.txt', 'utf8', (err, data) => {

if (err) return console.error(err);

console.log(data);

});

**b. Promises:**

Use .catch():

someAsyncFunction()

.then(result => console.log(result))

.catch(error => console.error(error));

**c. async/await:**

Use try...catch:

try {

const data = await fetchData();

} catch (err) {

console.error("Error occurred:", err);

}

**📝 Summary**

| **Concept** | **Use Case** |
| --- | --- |
| 🔁 Callbacks | Traditional way to handle async code |
| 📜 Promises | Modern way to chain async tasks |
| ⏱️ async/await | Cleaner, easier-to-read async code |
| 🔄 Event Loop | Powers non-blocking execution |
| ❗ Error Handling | Always handle async errors properly |

**🔵 SECTION 1: Node.js – Basics to Advanced**

**📘 4. HTTP & Server Basics**

**✅ 1. Creating a Basic HTTP Server**

Node.js comes with a built-in http module that allows us to create web servers without any external library.

**Example:**

const http = require('http');

const server = http.createServer((req, res) => {

res.end('Hello from Node.js Server!');

});

server.listen(3000, () => {

console.log('Server running at http://localhost:3000');

});

✅ This starts a server on port 3000 and sends a plain text response.

**✅ 2. Handling Requests and Responses**

You can read data from the request (req) and write to the response (res).

**Example: Basic routing**

const server = http.createServer((req, res) => {

if (req.url === '/') {

res.writeHead(200, {'Content-Type': 'text/plain'});

res.end('Home Page');

} else if (req.url === '/about') {

res.writeHead(200, {'Content-Type': 'text/plain'});

res.end('About Page');

} else {

res.writeHead(404, {'Content-Type': 'text/plain'});

res.end('404 Not Found');

}

});

🧠 Use req.url to route, and res.writeHead() to send headers (like content type, status code).

**✅ 3. Status Codes**

HTTP status codes are 3-digit numbers that indicate the result of an HTTP request.

| **Code** | **Meaning** |
| --- | --- |
| 200 | OK |
| 201 | Created |
| 400 | Bad Request |
| 401 | Unauthorized |
| 403 | Forbidden |
| 404 | Not Found |
| 500 | Internal Server Error |

**Example:**

res.writeHead(200, { 'Content-Type': 'text/plain' });

**✅ 4. Serving HTML and JSON**

**a. Serving HTML:**

const fs = require('fs');

const server = http.createServer((req, res) => {

if (req.url === '/') {

fs.readFile('index.html', (err, data) => {

res.writeHead(200, { 'Content-Type': 'text/html' });

res.end(data);

});

}

});

📁 Place index.html in the same folder.

**b. Serving JSON:**

const server = http.createServer((req, res) => {

if (req.url === '/api') {

const data = { name: 'John', age: 25 };

res.writeHead(200, { 'Content-Type': 'application/json' });

res.end(JSON.stringify(data));

}

});

**🧠 Summary**

| **Feature** | **Description** |
| --- | --- |
| http.createServer() | Creates a basic web server |
| req & res | Handle incoming request & send back response |
| writeHead() | Set HTTP status code and headers |
| Serve HTML | Use fs.readFile() and send with 'text/html' |
| Serve JSON | Use JSON.stringify() and 'application/json' |

**🔵 SECTION 1: Node.js – Basics to Advanced**

**📘 5. npm & Package Management**

**✅ 1. npm – Node Package Manager**

* **npm** is the default package manager for Node.js.
* Helps you **install**, **manage**, and **share** JavaScript packages/libraries.
* Comes bundled with Node.js.

Check version:

npm -v

**✅ 2. package.json & package-lock.json**

**a. package.json**

It is the **metadata file** of your Node.js project.

Generated using:

npm init *# Interactive*

npm init -y *# Default options*

**Example:**

{

"name": "my-app",

"version": "1.0.0",

"scripts": {

"start": "node app.js"

},

"dependencies": {

"express": "^4.18.2"

}

}

🔸 Contains info about project, scripts, dependencies, etc.

**b. package-lock.json**

* Automatically created when installing packages.
* Locks the exact version of every installed package and its sub-dependencies.
* Ensures **same versions** are installed across environments.

**✅ 3. Installing Packages**

**a. Install a package:**

npm install express

Installs package and adds it to dependencies in package.json.

**b. Install a specific version:**

npm install express@4.17.1

**c. Install globally:**

npm install -g nodemon

📦 Global packages are available from anywhere on your system.

**✅ 4. Dev Dependencies vs Prod Dependencies**

| **Type** | **Command** | **Stored In** | **Use For** |
| --- | --- | --- | --- |
| Dependencies | npm install <pkg> | "dependencies" | Required in production |
| Dev Dependencies | npm install <pkg> --save-dev | "devDependencies" | Only needed during development (e.g., testing, linting) |

**Example:**

npm install nodemon --save-dev

**✅ 5. Versioning (^, ~, etc.)**

| **Symbol** | **Meaning** | **Example** | **Version Range** |
| --- | --- | --- | --- |
| ^ | Accept latest minor/patch | ^1.2.3 | >=1.2.3 <2.0.0 |
| ~ | Accept latest patch | ~1.2.3 | >=1.2.3 <1.3.0 |
| No symbol | Exact version | 1.2.3 | =1.2.3 |

**Best Practice:**

* Use ^ for libraries you trust to follow **semver** (Semantic Versioning).
* Use exact versions in production if stability is critical.

**🧠 Summary**

| **Topic** | **Key Point** |
| --- | --- |
| package.json | Project metadata and dependency list |
| package-lock.json | Locks exact versions of installed packages |
| npm install | Installs dependencies |
| Dev vs Prod | --save-dev adds dev-only packages |
| Versioning | ^ = minor/patch updates, ~ = patch only |

**🔵 SECTION 1: Node.js – Basics to Advanced**

**📘 6. Debugging & Logging**

**✅ 1. console.log() – Basic Debugging**

The simplest way to debug:

console.log("Server started");

console.log({ req });

✅ Use it for:

* Quick variable inspection
* Printing function call flow
* Checking API responses

**✅ 2. debug Module**

Better alternative to console.log() for production-safe debugging.

**Install:**

npm install debug

**Use:**

const debug = require('debug')('app:server');

debug('Starting server...');

app.listen(3000, () => {

debug('Listening on port 3000');

});

🔹 Toggle output by setting the environment variable:

DEBUG=app:\* node app.js

**✅ 3. node inspect – Built-in Debugger**

Start your Node app in debug mode:

node inspect app.js

Use commands like:

* c → continue
* n → next line
* s → step into
* repl → open interactive console

🔹 Open Chrome DevTools:

node --inspect app.js

Then go to: chrome://inspect

**✅ 4. VSCode Debugger**

1. Add a breakpoint in VSCode by clicking beside the line number.
2. Go to the **Run and Debug** panel.
3. Create launch.json or click **“Run”** with “Node.js” config.

**Sample launch.json:**

{

"type": "node",

"request": "launch",

"name": "Debug App",

"program": "${workspaceFolder}/app.js"

}

✅ Benefits:

* Visual breakpoints
* Call stack and variable watch
* Step-in, step-over execution

**✅ 5. Logging Best Practices**

| **Practice** | **Description** |
| --- | --- |
| ✅ Use logging libraries | Use winston, morgan, or pino for structured logging |
| ✅ Separate logs | Separate logs for dev and production |
| ✅ Use log levels | e.g., info, warn, error, debug |
| ❌ Avoid logs in production | Avoid sensitive info in production logs |
| ✅ Store logs | Use files or external services (Logstash, Cloudwatch) |

**🔸 Exercises**

**✅ 1. Create a File Reader**

const fs = require('fs');

fs.readFile('data.txt', 'utf8', (err, data) => {

if (err) return console.error(err);

console.log(data);

});

**✅ 2. Basic HTTP Server**

const http = require('http');

const server = http.createServer((req, res) => {

res.end('Server running...');

});

server.listen(3000, () => console.log('Listening on 3000'));

**🔸 Mini Project: Simple File Upload Server**

**1. Setup**

Install express and multer:

npm install express multer

**2. upload.js**

const express = require('express');

const multer = require('multer');

const app = express();

const storage = multer.diskStorage({

destination: './uploads/',

filename: (req, file, cb) => {

cb(null, file.originalname);

}

});

const upload = multer({ storage });

app.post('/upload', upload.single('file'), (req, res) => {

res.send('File uploaded');

});

app.listen(3000, () => console.log('Server on 3000'));

**3. Use Postman or frontend form to test /upload.**

**🔸 Interview Question**

**❓ What is event-driven programming in Node.js?**

**Answer:** Node.js uses **event-driven architecture**, meaning the flow of the program is determined by **events** (like requests, responses, I/O completion) and **callbacks** that respond to those events.

The core component is the **Event Loop**, which continuously listens for and dispatches events. This allows Node.js to handle **non-blocking I/O** and scale efficiently.

**Example:**

const EventEmitter = require('events');

const emitter = new EventEmitter();

emitter.on('message', (data) => {

console.log('Received:', data);

});

emitter.emit('message', 'Hello Event!');

**✅ Summary**

| **Tool** | **Use** |
| --- | --- |
| console.log() | Quick checks |
| debug | Namespace-based logging |
| node inspect | CLI debugging |
| VSCode Debugger | Visual breakpoints and step debugging |
| Logging tools | Use winston, pino, etc. for real projects |

**🟠 SECTION 2: Express.js – Basics to Advanced**

**📘 1. What is Express?**

**✅ What is Express?**

* **Express.js** is a **minimal and flexible Node.js web application framework** that provides a robust set of features to build web and mobile applications.
* It’s a wrapper over Node’s native HTTP module—makes server-side development **faster and cleaner**.

🔸 Think of Express as the **“backend framework”** for Node.js, similar to how React is for the frontend.

**✅ Why Use Express with Node?**

| **Feature** | **Benefit** |
| --- | --- |
| 🚀 Simplicity | Clean and readable syntax |
| 🛠 Middleware Support | Easily add custom and third-party functions to the request/response cycle |
| 📦 Routing | Built-in support for URL routing |
| 📁 Static Files | Built-in way to serve static files |
| 🌐 REST APIs | Ideal for building RESTful APIs quickly |

🔥 **Real-world use:** Used in MERN stack (MongoDB, Express, React, Node.js)

**✅ Setting Up an Express Project**

**1. Initialize a new project**

mkdir express-app

cd express-app

npm init -y

**2. Install Express**

npm install express

**✅ Basic Express Server**

**Create index.js:**

const express = require('express');

const app = express();

*// Route*

app.get('/', (req, res) => {

res.send('Hello from Express!');

});

*// Start server*

app.listen(3000, () => {

console.log('Server running at http://localhost:3000');

});

**Run the server:**

node index.js

🟢 Open http://localhost:3000 in browser to see your response.

**✅ Explanation:**

| **Code** | **Purpose** |
| --- | --- |
| express() | Initializes the app |
| app.get() | Handles GET request at route / |
| res.send() | Sends back a response to the client |
| app.listen() | Starts the server on a given port |

**🧠 Pro Tip:**

Use nodemon for auto-reload on changes:

npm install --save-dev nodemon

npx nodemon index.js

**🟠 SECTION 2: Express.js – Basics to Advanced**

**📘 2. Express Routing**

**✅ What is Routing?**

Routing defines how your app responds to **client requests** to a particular **URL path** and **HTTP method** (GET, POST, etc.).

**✅ 1. HTTP Methods in Express**

| **Method** | **Purpose** |
| --- | --- |
| GET | Read/Retrieve data |
| POST | Create new data |
| PUT | Update existing data |
| DELETE | Delete data |

**🔹 Basic Syntax:**

app.get('/path', (req, res) => { ... });

app.post('/path', (req, res) => { ... });

app.put('/path', (req, res) => { ... });

app.delete('/path', (req, res) => { ... });

**✅ Example:**

const express = require('express');

const app = express();

app.use(express.json()); *// to parse JSON request body*

app.get('/users', (req, res) => {

res.send('Get all users');

});

app.post('/users', (req, res) => {

res.send('Create a user');

});

app.put('/users/:id', (req, res) => {

res.send(`Update user with ID: ${req.params.id}`);

});

app.delete('/users/:id', (req, res) => {

res.send(`Delete user with ID: ${req.params.id}`);

});

app.listen(3000);

**✅ 2. Route Parameters**

Used to capture values from the URL (dynamic routing).

**Example:**

app.get('/product/:id', (req, res) => {

res.send(`Product ID: ${req.params.id}`);

});

🔹 :id is a route parameter. You can access it via req.params.id.

**✅ 3. Query Strings**

Passed in the URL after ? to filter or sort data.

**URL:** http://localhost:3000/search?term=node&page=2

**Route Example:**

app.get('/search', (req, res) => {

const { term, page } = req.query;

res.send(`Search term: ${term}, Page: ${page}`);

});

✅ Access query string values via req.query.

**✅ 4. Express Router – Modular Routing**

Helps organize routes into separate files.

**Step 1: Create routes/userRoutes.js**

const express = require('express');

const router = express.Router();

router.get('/', (req, res) => {

res.send('All users');

});

router.post('/', (req, res) => {

res.send('Create user');

});

module.exports = router;

**Step 2: Import and use it in app.js**

const express = require('express');

const app = express();

const userRoutes = require('./routes/userRoutes');

app.use('/users', userRoutes);

app.listen(3000);

**🧠 Summary**

| **Concept** | **Key Point** |
| --- | --- |
| HTTP Methods | Define action: GET, POST, PUT, DELETE |
| Route Params | :id for dynamic values |
| Query Strings | req.query for filters/sorting |
| Express Router | Organize routes modularly |

**🟠 SECTION 2: Express.js – Basics to Advanced**

**📘 3. Middleware in Express**

**✅ What is Middleware?**

* **Middleware** functions are functions that have access to the **request** and **response** objects, and the **next** function.
* They are executed in sequence during the request-response cycle.

**Basic Structure:**

(req, res, next) => {

*// Logic here*

next(); *// Pass control to next middleware*

}

**✅ 1. Built-in Middleware**

| **Middleware** | **Purpose** |
| --- | --- |
| express.json() | Parses incoming JSON payloads |
| express.urlencoded() | Parses URL-encoded form data |

**Example:**

const express = require('express');

const app = express();

app.use(express.json()); *// to parse JSON*

app.use(express.urlencoded({ extended: true })); *// for form data*

🔸 Always put .use() before routes.

**✅ 2. Custom Middleware**

You can write your own middleware to log, validate, or manipulate requests.

**Example: Logger Middleware:**

app.use((req, res, next) => {

console.log(`${req.method} ${req.url}`);

next(); *// call next middleware*

});

🔹 Custom middleware must call next() to proceed.

**✅ 3. Error-handling Middleware**

Express identifies error middleware by the **4 arguments**:

(err, req, res, next) => {

console.error(err.message);

res.status(500).send('Something went wrong!');

}

**Use case:**

app.use((err, req, res, next) => {

res.status(500).json({ error: err.message });

});

🔸 Always define **after all routes**.

**✅ 4. Using next()**

| **Situation** | **Use** |
| --- | --- |
| Normal middleware | next() moves to next handler |
| In error | next(err) passes error to error-handling middleware |

**Example with error:**

app.use((req, res, next) => {

const err = new Error('Not found');

next(err); *// Go to error middleware*

});

**✅ Middleware Execution Flow Example**

app.use((req, res, next) => {

console.log('Middleware 1');

next();

});

app.use((req, res, next) => {

console.log('Middleware 2');

next();

});

app.get('/', (req, res) => {

res.send('Final handler');

});

**Console output:**

Middleware 1

Middleware 2

Final handler

**🧠 Summary**

| **Type** | **Purpose** |
| --- | --- |
| Built-in | express.json(), express.urlencoded() |
| Custom | Logging, validation, auth |
| Error-handling | Handle thrown/caught errors |
| next() | Moves request through middleware stack |

**🟠 SECTION 2: Express.js – Basics to Advanced**

**📘 4. Templates & Static Files**

**✅ 1. Serving Static Files in Express**

Static files include images, CSS, JS, HTML—files that don’t change dynamically.

**Step 1: Create a public/ folder:**

project/

├── public/

│ ├── index.html

│ ├── style.css

│ └── script.js

**Step 2: Serve static files:**

const express = require('express');

const app = express();

app.use(express.static('public'));

app.listen(3000);

🟢 Now you can access files like:

* http://localhost:3000/index.html
* http://localhost:3000/style.css

**✅ 2. What is a View Engine?**

* A **View Engine** lets you write HTML templates that include **dynamic data** using logic (loops, conditionals, etc.).
* Popular engines: **EJS**, Pug, Handlebars.

**✅ Using EJS with Express**

**Step 1: Install EJS**

npm install ejs

**Step 2: Set the view engine**

app.set('view engine', 'ejs');

**Step 3: Create a views/ folder with .ejs files**

**views/home.ejs**

<!DOCTYPE html>

<html>

<head>

<title><%= title %></title>

</head>

<body>

<h1>Welcome, <%= user %>!</h1>

</body>

</html>

**Step 4: Render a template from route**

app.get('/', (req, res) => {

res.render('home', { title: 'Home Page', user: 'John' });

});

✅ res.render() injects dynamic values into your .ejs template.

**✅ EJS Syntax Summary**

| **Syntax** | **Purpose** |
| --- | --- |
| <%= variable %> | Output value |
| <% if () {} %> | Conditional rendering |
| <% for (...) { } %> | Loop over array |

**🧠 Summary**

| **Feature** | **Code** |
| --- | --- |
| Static files | app.use(express.static('public')) |
| Set view engine | app.set('view engine', 'ejs') |
| Render EJS | res.render('file', data) |

**🟠 SECTION 2: Express.js – Basics to Advanced**

**📘 5. REST API with Express**

**✅ What is a REST API?**

A **REST API** (Representational State Transfer) is an architecture that uses standard HTTP methods to perform **CRUD operations**:

| **Method** | **Action** | **Example URL** |
| --- | --- | --- |
| GET | Read | /api/users |
| POST | Create | /api/users |
| PUT | Update | /api/users/:id |
| DELETE | Delete | /api/users/:id |

**✅ Basic Express REST API Example**

const express = require('express');

const app = express();

app.use(express.json());

let users = [

{ id: 1, name: 'Alice' },

{ id: 2, name: 'Bob' }

];

*// GET - Read all users*

app.get('/api/users', (req, res) => {

res.json(users);

});

*// POST - Create user*

app.post('/api/users', (req, res) => {

const { name } = req.body;

const newUser = { id: users.length + 1, name };

users.push(newUser);

res.status(201).json(newUser);

});

*// PUT - Update user*

app.put('/api/users/:id', (req, res) => {

const user = users.find(u => u.id == req.params.id);

if (!user) return res.status(404).json({ error: 'User not found' });

user.name = req.body.name;

res.json(user);

});

*// DELETE - Delete user*

app.delete('/api/users/:id', (req, res) => {

users = users.filter(u => u.id != req.params.id);

res.json({ message: 'User deleted' });

});

app.listen(3000, () => console.log('API running on port 3000'));

**✅ HTTP Status Codes (Important!)**

| **Code** | **Meaning** | **Use in REST** |
| --- | --- | --- |
| 200 | OK | GET, PUT, DELETE success |
| 201 | Created | POST success |
| 400 | Bad Request | Validation failed |
| 404 | Not Found | Resource not found |
| 500 | Internal Server Error | Unexpected issue |

**✅ Input Validation Example**

app.post('/api/users', (req, res) => {

if (!req.body.name || req.body.name.length < 3) {

return res.status(400).json({ error: 'Name must be at least 3 characters' });

}

const user = { id: users.length + 1, name: req.body.name };

users.push(user);

res.status(201).json(user);

});

**✅ Using Postman to Test APIs**

1. **Download Postman:** <https://www.postman.com/downloads/>
2. **Start your Express server**
3. In Postman:
   * Choose method: GET, POST, PUT, DELETE
   * Set URL: http://localhost:3000/api/users
   * For POST/PUT, go to **Body → raw → JSON**, and input:
   * { "name": "Charlie" }
   * Click **Send** and view response.

**✅ REST API Summary**

| **Task** | **Method** | **Endpoint** | **Code Example** |
| --- | --- | --- | --- |
| Read users | GET | /api/users | app.get() |
| Add user | POST | /api/users | app.post() |
| Update user | PUT | /api/users/:id | app.put() |
| Delete user | DELETE | /api/users/:id | app.delete() |

**🟠 SECTION 2: Express.js – Basics to Advanced**

**📘 6. Express + MongoDB Integration**

**✅ 1. Connecting MongoDB with Mongoose**

**Mongoose** is an ODM (Object Data Modeling) library for MongoDB and Node.js. It simplifies schema creation, validation, and interaction with MongoDB.

**🔹 Install Mongoose**

npm install mongoose

**🔹 Connect to MongoDB**

const mongoose = require('mongoose');

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

useNewUrlParser: true,

useUnifiedTopology: true

})

.then(() => console.log('MongoDB connected'))

.catch(err => console.error('MongoDB connection error:', err));

**✅ 2. Create Mongoose Model**

**models/User.js**

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

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

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

age: { type: Number }

});

module.exports = mongoose.model('User', userSchema);

**✅ 3. Routes + CRUD Operations with MongoDB**

**routes/userRoutes.js**

const express = require('express');

const router = express.Router();

const User = require('../models/User');

*// GET all users*

router.get('/', async (req, res) => {

const users = await User.find();

res.json(users);

});

*// POST new user*

router.post('/', async (req, res) => {

try {

const user = new User(req.body);

await user.save();

res.status(201).json(user);

} catch (err) {

res.status(400).json({ error: err.message });

}

});

*// PUT update user*

router.put('/:id', async (req, res) => {

try {

const user = await User.findByIdAndUpdate(req.params.id, req.body, { new: true });

res.json(user);

} catch (err) {

res.status(400).json({ error: err.message });

}

});

*// DELETE user*

router.delete('/:id', async (req, res) => {

await User.findByIdAndDelete(req.params.id);

res.json({ message: 'User deleted' });

});

module.exports = router;

**✅ 4. Register Router in Express App**

**app.js**

const express = require('express');

const mongoose = require('mongoose');

const userRoutes = require('./routes/userRoutes');

const app = express();

app.use(express.json());

*// Connect DB*

mongoose.connect('mongodb://127.0.0.1:27017/expressdb')

.then(() => console.log('MongoDB connected'));

*// Use Routes*

app.use('/api/users', userRoutes);

app.listen(3000, () => console.log('Server running'));

**✅ 5. MVC Pattern in Express Apps**

| **Layer** | **Responsibility** | **Folder** |
| --- | --- | --- |
| **Model** | MongoDB Schema/Logic | /models |
| **View** | (Optional) EJS, etc. | /views |
| **Controller** | Request handling logic | /controllers |
| **Route** | Route definitions | /routes |
| **App** | Entry file | app.js |

**Example Folder Structure:**

/models/User.js

/routes/userRoutes.js

/controllers/userController.js

/app.js

🔸 MVC keeps your code **organized and scalable**.

**🧠 Summary**

| **Feature** | **Code / Tool** |
| --- | --- |
| ODM | mongoose |
| Connect to DB | mongoose.connect() |
| Schema/Model | mongoose.Schema() |
| CRUD | find(), save(), findByIdAndUpdate(), findByIdAndDelete() |
| Structure | MVC (Model-View-Controller) |

**🟠 SECTION 2: Express.js – Basics to Advanced**

**📘 7. Authentication (Intro)**

**✅ What is Authentication?**

Authentication is the process of **verifying a user’s identity** (e.g., login with username/email and password).

In Express apps, common authentication methods include:

* **Sessions & Cookies**
* **JWT (JSON Web Tokens)**

**✅ 1. Sessions & Cookies**

**🔹 What are Cookies?**

* Cookies are small pieces of data stored on the **client (browser)**.
* They store session IDs or tokens used for authentication.

**🔹 What are Sessions?**

* Sessions store data **on the server**, tied to a session ID (stored in a cookie on the client).
* When a user logs in, a **session** is created and linked to that user.

**🔸 How to Use Sessions in Express**

**🧩 Install express-session**

npm install express-session

**🔐 Setup Example**

const session = require('express-session');

app.use(session({

secret: 'your-secret-key',

resave: false,

saveUninitialized: true,

cookie: { secure: false } *// Use `true` with HTTPS*

}));

**🔐 Create & Use Sessions**

*// Login route*

app.post('/login', (req, res) => {

const { username } = req.body;

req.session.user = username;

res.send('Logged in');

});

*// Protected route*

app.get('/dashboard', (req, res) => {

if (req.session.user) {

res.send(`Welcome ${req.session.user}`);

} else {

res.status(401).send('Unauthorized');

}

});

**🔓 Logout**

app.get('/logout', (req, res) => {

req.session.destroy();

res.send('Logged out');

});

**✅ 2. JWT (JSON Web Tokens)**

JWT is a **stateless** authentication method using tokens instead of sessions.

**🔹 How JWT Works:**

1. User logs in.
2. Server creates a token (signed with secret key) and sends it to client.
3. Client stores it (localStorage or cookie).
4. On each request, client sends token in the Authorization header.
5. Server verifies token.

**🔸 JWT Example with jsonwebtoken**

**🧩 Install:**

npm install jsonwebtoken

**🔐 Generate Token**

const jwt = require('jsonwebtoken');

app.post('/login', (req, res) => {

const user = { id: 1, name: 'John' };

const token = jwt.sign(user, 'secret123', { expiresIn: '1h' });

res.json({ token });

});

**🧾 Middleware to Verify Token**

const authenticate = (req, res, next) => {

const authHeader = req.headers['authorization'];

const token = authHeader?.split(' ')[1];

if (!token) return res.sendStatus(401);

jwt.verify(token, 'secret123', (err, user) => {

if (err) return res.sendStatus(403);

req.user = user;

next();

});

};

*// Protected route*

app.get('/profile', authenticate, (req, res) => {

res.json({ message: 'Welcome to profile', user: req.user });

});

**🔐 Sessions vs JWT – Comparison**

| **Feature** | **Sessions** | **JWT** |
| --- | --- | --- |
| Storage | Server | Client (token) |
| Stateless | ❌ No | ✅ Yes |
| Scalable | ❌ Less | ✅ More |
| Security | Secure with HTTPS | Needs token protection |
| Use case | Web apps (server-rendered) | APIs (RESTful) |

**🧠 Summary**

| **Concept** | **Notes** |
| --- | --- |
| Cookie | Stored on browser |
| Session | Stored on server |
| JWT | Token stored on client; verified by server |
| Package | express-session, jsonwebtoken |

**🟠 SECTION 2: Express.js – Basics to Advanced**

**📘 🔸 Exercises, Mini Project & Interview Questions**

**✅ 🔸 Exercises**

**1. Simple Routing**

const express = require('express');

const app = express();

app.get('/', (req, res) => res.send('Home Page'));

app.get('/about', (req, res) => res.send('About Page'));

app.get('/contact', (req, res) => res.send('Contact Page'));

app.listen(3000, () => console.log('Server running on port 3000'));

**2. Custom Middleware**

const logger = (req, res, next) => {

console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);

next(); *// continue to next middleware/route*

};

app.use(logger);

Use with a route:

app.get('/profile', (req, res) => {

res.send('Profile Page');

});

**✅ 🔸 Mini Project – To-Do App API (No DB)**

**🧩 File: todoApp.js**

const express = require('express');

const app = express();

app.use(express.json());

let todos = []; *// in-memory array*

*// Get all todos*

app.get('/todos', (req, res) => {

res.json(todos);

});

*// Add a new todo*

app.post('/todos', (req, res) => {

const todo = { id: Date.now(), task: req.body.task, done: false };

todos.push(todo);

res.status(201).json(todo);

});

*// Update a todo*

app.put('/todos/:id', (req, res) => {

const todo = todos.find(t => t.id == req.params.id);

if (!todo) return res.status(404).json({ error: 'Not found' });

todo.task = req.body.task ?? todo.task;

todo.done = req.body.done ?? todo.done;

res.json(todo);

});

*// Delete a todo*

app.delete('/todos/:id', (req, res) => {

todos = todos.filter(t => t.id != req.params.id);

res.json({ message: 'Deleted' });

});

app.listen(3000, () => console.log('To-Do API running'));

🧪 Test with Postman or browser:

* GET /todos
* POST /todos with JSON { "task": "Buy milk" }
* PUT /todos/:id
* DELETE /todos/:id

**✅ 🔸 Interview Question**

**❓ What’s the difference between middleware and a route handler?**

| **Feature** | **Middleware** | **Route Handler** |
| --- | --- | --- |
| Purpose | Intercept & modify requests/responses | Respond to a specific HTTP request |
| Executes via | app.use() or before route | app.get(), app.post() etc. |
| Next() usage | Must call next() to continue chain | Ends request by sending a response |
| Example | Logger, auth checker, body parser | /api/users – returns users list |

**Example:**

*// Middleware*

app.use((req, res, next) => {

console.log('Middleware ran');

next();

});

*// Route handler*

app.get('/home', (req, res) => {

res.send('Welcome Home');

});

**🟢 SECTION 3: MongoDB + Mongoose – Basics to Advanced**

**📘 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) |

**🟢 SECTION 3: MongoDB + Mongoose – Basics to Advanced**

**📘 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 |

**🟢 SECTION 3: MongoDB + Mongoose – Basics to Advanced**

**📘 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 |

**🟢 SECTION 3: MongoDB + Mongoose – Basics to Advanced**

**📘 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 |

**🟢 SECTION 3: MongoDB + Mongoose – Basics to Advanced**

**📘 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 |

**🟢 SECTION 3: MongoDB + Mongoose – Basics to Advanced**

**📘 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 |

**🟢 SECTION 3: MongoDB + Mongoose – Basics to Advanced**

**📘 Practice & Application**

**🔸 Exercises**

1. **Insert & Update Data**
   * 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.