Node.js, Express.js, MongoDB 150+ Theory Interview Q&A, 10+ Tasks

This page is your all-in-one guide to preparing for interviews on **Node.js**, **Express.js**, and **MongoDB**. It includes **150+ theory questions and 10 + coding tasks with solutions** to help you understand and practice these technologies. Everything is explained in simple words to make it easy for you to learn, even if the concepts are advanced.

What's Included:

- **Theory Questions**: Covering key concepts from Node.js, Express.js, and MongoDB in simple terms.
- **Coding Tasks**: Practical challenges with detailed solutions to improve your problem-solving skills.

Topics Covered:

Node.js Topics

- Reading and Writing Files Asynchronously
- Event Emitters
- Using mongodump and mongorestore for Backups
- Streams in Node.js
- Buffers in Node.js
- Child Processes and Worker Threads
- File System Operations

Express.js Topics

- Creating RESTful APIs
- Middleware Implementation
- File Uploads with Multer
- Authentication and Authorization (bcrypt, JWT, RBAC)
- Rate Limiting with express-rate-limit
- Pagination and Query Parameters
- Soft Deletion for Records
- Real-Time Notifications with SSE
- Validation Middleware
- Error Handling Middleware
- CORS Handling
- Search with MongoDB Text Indexing

MongoDB Topics

- Aggregation Framework
- Indexing (Single-field, Compound, Text, Hashed)

- Replica Sets and Sharding
- GridFS (File Uploads and Retrieval)
- Soft Delete with Logical Flags
- Point-in-Time Recovery
- Change Streams for Real-Time Updates
- Backup and Restore (Physical and Logical)
- Bulk Insert Operations
- Working with Time-Series Data

Node.js, Express.js and MongoDB 150+ Theory Interview Questions

Introduction to Node.js:

Q1. What is Node.js?

Answer:
Node.js is an open-source, cross-platform runtime environment for executing JavaScript code outside of a browser. It is built on Google Chrome's V8 JavaScript engine and is widely used for building server-side and network applications. Key points include:

1. **JavaScript Runtime:** It allows JavaScript to run on the server, enabling full-stack development using a single language.
2. **Non-Blocking I/O Model:** Supports handling multiple requests simultaneously, making it efficient and scalable.
3. **Event-Driven Architecture:** Its asynchronous programming model ensures that operations do not block the execution of other code.
4. **Platform Independence:** Node.js runs on various operating systems, such as Windows, macOS, and Linux.
5. **Rich Ecosystem:** Comes with NPM (Node Package Manager) for easy management of libraries and modules.

Q2. What are the main features of Node.js?

Answer:
Node.js offers a range of features that make it popular for developers. Key features include:

1. **Fast Execution:** Built on the V8 engine, Node.js executes JavaScript code quickly.
2. **Asynchronous Processing:** Non-blocking I/O ensures better performance by processing multiple requests simultaneously.
3. **Single Programming Language:** Developers use JavaScript for both front-end and back-end, simplifying development.
4. **Scalable Applications:** Its single-threaded architecture with an event loop supports high scalability for real-time apps.
5. **NPM Library Support:** Provides access to thousands of pre-built modules for faster and easier development.
6. **Event-Driven:** Node.js uses an event-driven model, making it suitable for real-time applications like chat and gaming apps.

Q3. What are some common uses of Node.js?

Answer:
Node.js is versatile and widely used in various domains. Common use cases include:

1. **Real-Time Applications:** Ideal for chat applications, gaming servers, and live collaboration tools due to its event-driven model.
2. **API Development:** Perfect for building RESTful APIs and microservices.
3. **Single Page Applications (SPA):** Provides a seamless experience by reducing server-side load.

4. **Streaming Applications:** Efficient in handling large data streams, such as video or audio streaming platforms.
5. **Server-Side Proxy:** Can handle multiple service requests and act as a proxy for other servers.

Q4. Why is Node.js asynchronous?

Answer:
Node.js follows an asynchronous programming model to improve performance and scalability. Reasons include:

1. **Non-Blocking Nature:** Operations like file reading or database queries don't block the execution of other tasks.
2. **Efficient Resource Usage:** Reduces waiting time, allowing more operations to run concurrently.
3. **Event Loop Mechanism:** Uses a single-threaded event loop to manage multiple tasks without creating multiple threads.
4. **Scalability:** Suitable for high-traffic applications due to its ability to handle numerous requests simultaneously.
5. **Faster Execution:** Promotes faster execution of operations, especially in real-time applications.

Q5. What are the advantages of using Node.js?

Answer:
Node.js offers several advantages that make it a preferred choice for developers:

1. **High Performance:** Powered by the V8 engine for quick execution of JavaScript.
2. **Easy Scalability:** Handles thousands of connections simultaneously with its event-driven model.
3. **Unified Language:** JavaScript is used for both server and client-side development.
4. **Rich Package Ecosystem:** NPM offers numerous reusable modules, reducing development time.
5. **Community Support:** Active and growing community ensures regular updates and support.
6. **Real-Time Capabilities:** Excellent for applications requiring real-time updates, such as chat and collaboration tools.

Core Modules:

Q6. What are Node.js core modules, and why are they important?

Answer:
Node.js core modules are built-in modules provided by Node.js for various functionalities.

1. **Built-In Availability:** Pre-installed and ready to use without any additional setup.
2. **Optimized Performance:** Written in C++ and integrated tightly with Node.js for efficiency.

3. **Direct Integration:** Used with require() without specifying a path.
4. **Wide Use Cases:** Handle tasks like file I/O, HTTP servers, streams, and more.
5. **Foundation for Applications:** Core modules are the building blocks for creating scalable and efficient applications.
6. **Reduced Dependencies:** No need for external libraries, ensuring smaller project sizes.
7. **Cross-Platform Compatibility:** Work seamlessly across operating systems like Windows, Linux, and macOS.
8. **Support for Asynchronous and Synchronous APIs:** Developers can choose the suitable method based on their use case.

Q7. Name and briefly describe five common core modules in Node.js.

Answer:

1. **fs (File System):**
   Handles file operations like reading, writing, and deleting files. Supports both synchronous and asynchronous methods.
2. **http:**
   Used to create web servers and handle HTTP requests and responses.
3. **path:**
   Provides utilities to work with file and directory paths.
4. **os:**
   Offers information about the operating system, such as CPU usage and memory status.
5. **events:**
   Implements the event-driven programming model, allowing objects to emit and listen for events.

Q8. Explain the http module and its use in creating an HTTP server.

Answer:
The http module enables the creation of HTTP servers and handling requests and responses.

1. **Server Creation:** Use http.createServer to start a web server.
2. **Request Handling:** Manage incoming requests with the request object (req).
3. **Response Control:** Send responses using the response object (res).
4. **Header Management:** Customize HTTP headers using res.writeHead().
5. **Integration with APIs:** Build RESTful APIs by parsing URL parameters and query strings.
6. **Event Listeners:** Attach events like request, connection, and close.
7. **Middleware Integration:** Combine with frameworks like Express.js for extended functionality.

```
const http = require("http");
const server = http.createServer((req, res) => {
  res.writeHead(200, { "Content-Type": "text/plain" });
  res.end("Hello World");
});
server.listen(3000, () => console.log("Server is running on port 3000"));
```

Node.js, Express.js and MongoDB 150+ Theory Interview Questions

Q9. Explain the difference between fs.readFile() and fs.readFileSync().

Answer:

fs.readFile() (Asynchronous):

- Non-blocking and uses a callback to handle results.

```
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

fs.readFileSync() (Synchronous):

- Blocking and does not proceed until the file is read.

```
const data = fs.readFileSync("file.txt", "utf8");
console.log(data);
```

Use Cases:

- fs.readFile() is preferred for non-blocking operations.
- fs.readFileSync() is used when subsequent code depends on the file's content.

Performance Impact:

- fs.readFileSync() can slow down performance in applications handling multiple requests.

Error Handling:

- Both methods require error handling for robustness.

Q10. What is the purpose of the events module in Node.js?

Answer:
The events module provides the EventEmitter class for creating and managing events.

1. **Event Emission:** Emit custom events with emit() for asynchronous communication.
2. **Event Listening:** Use on() or addListener() to respond to events.
3. **Built-In Integration:** Core modules like http and fs internally use EventEmitter.
4. **Custom Event Handling:** Developers can define and handle custom events.
5. **Memory Management:** Use removeListener or removeAllListeners to avoid memory leaks.
6. **Event Hierarchy:** Supports chaining multiple event handlers.

```
const EventEmitter = require("events");
const emitter = new EventEmitter();
```

```
emitter.on("greet", () => console.log("Hello!"));
emitter.emit("greet");
```

Q11. Describe the os module and its functionalities.

Answer:
The os module provides operating system-related utilities and information.

1. **CPU Information:** Retrieve CPU details using os.cpus().
2. **Memory Details:** Check total and free memory with os.totalmem() and os.freemem().
3. **System Uptime:** Get the system uptime in seconds using os.uptime().
4. **Home Directory:** Access the current user's home directory with os.homedir().
5. **Operating System Name:** Identify the OS type with os.type().
6. **Network Interfaces:** Get information about network interfaces with os.networkInterfaces().

Q12. What are streams in Node.js, and why are they used?

Answer:
Streams handle continuous data flows, making them efficient for large data processing.

1. **Readable Streams:** Used for input (e.g., fs.createReadStream).
2. **Writable Streams:** Handle output (e.g., fs.createWriteStream).
3. **Duplex Streams:** Support both reading and writing.
4. **Transform Streams:** Modify or transform data during reading or writing.
5. **Event-Driven:** Trigger events like data, end, and error.
6. **Memory Efficiency:** Process data in chunks rather than loading entire files into memory.

Q13. What are the main differences between synchronous and asynchronous methods in Node.js core modules?

Answer:

1. **Execution Blocking:** Synchronous methods block the thread until completion.
2. **Non-Blocking:** Asynchronous methods use callbacks or Promises to avoid blocking.
3. **Performance:** Asynchronous methods are preferred for I/O-heavy tasks.
4. **Error Handling:** Synchronous methods throw errors, while asynchronous methods return them in callbacks.
5. **Use Case:** Synchronous methods are used for simple scripts; asynchronous methods are better for servers.

NPM and Package Management:

Q14. What are the differences between local and global package installations in NPM?

Answer:

1. **Local Packages:**
    - Installed within a project's node_modules folder.

- Specific to the project and added to dependencies in package.json.
- Example: npm install lodash.

2. **Global Packages:**
   - Installed system-wide and accessible across all projects.
   - Useful for command-line tools and utilities.
   - Example: npm install -g nodemon.
3. **Command Behavior:**
   - Local installations require node_modules in the project directory.
   - Global installations make the package available via the command line.
4. **Scoping:**
   - Local packages are limited to the project, ensuring no conflicts.
   - Global packages may cause version conflicts across projects.

Q15. What is package.json, and what information does it contain?

Answer:
package.json is a metadata file that defines a Node.js project and its dependencies.

1. **Project Metadata:** Contains name, version, description, and author of the project.
2. **Dependencies:** Lists packages required for the project to run (dependencies) and for development purposes (devDependencies).
3. **Scripts:** Defines custom tasks like npm run build or npm run test.
4. **License:** Specifies the licensing details of the project.
5. **Keywords:** Includes keywords to make the project discoverable in the NPM registry.
6. **Engines:** Specifies compatible versions of Node.js.

Q16. How do you install, update, and uninstall a package using NPM?

Answer:

1. **Install a Package:**
   - Local: npm install package-name.
   - Global: npm install -g package-name.
2. **Update a Package:**
   - Specific Package: npm update package-name.
   - All Packages: npm update.
3. **Uninstall a Package:**
   - Local: npm uninstall package-name.
   - Global: npm uninstall -g package-name.
4. **Install Specific Version:**
   - Example: npm install package-name@1.2.3.
5. **Save Flags:**
   - --save: Adds the package to dependencies.
   - --save-dev: Adds the package to devDependencies

Q17. What are the differences between dependencies and devDependencies in package.json?

Answer:

1. **dependencies:**
    - Required for the application to run.
    - Installed using npm install --save.
    - Example: express for server handling.
2. **devDependencies:**
    - Required only during development.
    - Installed using npm install --save-dev.
    - Example: nodemon for auto-reloading during development.
3. **Production Use:**
    - npm install installs only dependencies in production environments.
    - Use npm install --only=dev to install devDependencies.
4. **Clarity in Codebase:**
    - Separates runtime and development requirements.

Q18. What is package-lock.json, and why is it important?

Answer:

1. **Dependency Tree:** Records the exact versions of installed packages.
2. **Reproducible Installs:** Ensures consistent installations across environments.
3. **Performance:** Speeds up installation by skipping version resolution.
4. **Nested Dependencies:** Locks the dependencies of dependencies.
5. **Version Tracking:** Prevents accidental upgrades of dependencies.
6. **Should Be Committed:** Recommended to include in version control for reliability.

Q19. What are NPM scripts, and how are they used?

Answer:
NPM scripts are commands defined in package.json to automate tasks.

1. **Custom Commands:** Run tasks like building, testing, or linting.
2. **Predefined Scripts:** Default scripts like start, test, and build.
3. **Execution:** Run using npm run script-name.
4. **Examples:**
    - "start": "node index.js".
    - "test": "jest".
5. **Shortcuts:** npm start directly runs the start script.

Q20. Explain the purpose of the node_modules folder.

Answer:

1. **Dependency Storage:** Contains all installed packages and their dependencies.
2. **Local Scope:** Specific to the project unless the package is installed globally.
3. **Automatic Updates:** Managed by NPM during installations and updates.
4. **Size:** Can grow large because it includes all nested dependencies.
5. **Not Version Controlled:** Typically excluded from version control (.gitignore).
6. **Recreated by NPM:** Automatically rebuilt using npm install if package.json and package-lock.json exist.

Node.js, Express.js and MongoDB 150+ Theory Interview Questions

Asynchronous Programming:

Q21. What is asynchronous programming, and how does Node.js implement it?

Answer:
Asynchronous programming is a non-blocking programming model that allows execution to continue while waiting for other tasks to complete.

1. **Non-Blocking:** Operations like file reading or API calls don't block the main thread.
2. **Event Loop:** Node.js uses an event loop to manage multiple tasks efficiently.
3. **Callbacks:** Functions are executed once an asynchronous task completes.
4. **Promises:** Provides a cleaner way to handle asynchronous operations with .then and .catch.
5. **Async/Await:** Simplifies writing asynchronous code that looks synchronous.
6. **Efficient Resource Usage:** Handles thousands of requests without creating multiple threads.

Q22. What is the event loop in Node.js, and why is it important?

Answer:
The event loop is the mechanism that allows Node.js to handle multiple tasks concurrently.

1. **Single-Threaded:** Manages all tasks on a single thread using callbacks.
2. **Non-Blocking I/O:** Handles tasks like file I/O or network requests asynchronously.
3. **Phases:** Includes phases like timers, I/O callbacks, idle/prepare, poll, check, and close callbacks.
4. **Queue System:** Tasks are queued and processed in order based on priority.
5. **Scalability:** Allows Node.js to handle high traffic with minimal resources.
6. **Heartbeat of Node.js:** The core that keeps the asynchronous model functional.

Q23. What are callbacks in Node.js, and what are their limitations?

Answer:
Callbacks are functions passed as arguments to be executed after an asynchronous task completes.

1. **Execution After Completion:** Called only after the task finishes.

```
const fs = require("fs");
fs.readFile("file.txt", "utf8", (err, data) => {
  if (err) console.error(err);
  else console.log(data);
});
```

2. **Error Handling:** Handles errors as the first argument in Node.js-style callbacks
3. **Callback Hell:** Nested callbacks make code difficult to read and maintain.
4. **No Return Value:** Can't directly return values from asynchronous operations.
5. **Replaced by Promises:** Modern alternatives like Promises and async/await overcome these limitations.

Q24. What is async/await, and how does it improve asynchronous code?

Answer:
async/await is a modern syntax for handling asynchronous operations in a synchronous style.

1. **Simplifies Code:** Makes asynchronous code easier to read and write.
2. **Requires Promises:** Works only with functions returning Promises.
3. **Error Handling:** Use try...catch for error management.

```
async function fetchData() {
  try {
    const data = await fetch("https://api.example.com");
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}
fetchData();
```

4. **Sequential Execution:** Allows step-by-step execution of asynchronous tasks.
5. **Cleaner Syntax:** Eliminates the need for .then chaining.

Q25. How does Node.js handle asynchronous errors?

Answer:
Node.js provides multiple ways to handle errors in asynchronous code.

1. **Callbacks:** Error passed as the first argument. Example: callback(err, result).
2. **Promises:** Use .catch() to handle errors.
3. **Async/Await:** Enclose await calls in try...catch blocks.
4. **Events:** Emit and listen for error events using the EventEmitter class.
5. **Global Error Handlers:** Handle uncaught exceptions with process.on('uncaughtException').
6. **Best Practices:** Always validate inputs and handle errors in every async operation.

Q26. What are some common use cases for asynchronous programming in Node.js?

Answer:

1. **File I/O:** Reading and writing files without blocking other tasks.
2. **Database Queries:** Fetching or updating data from databases like MongoDB.
3. **API Requests:** Handling HTTP requests and responses.
4. **Real-Time Applications:** Chat apps, gaming servers, or live collaboration tools.
5. **Event-Driven Tasks:** Using events to trigger actions.
6. **Third-Party Integrations:** Working with external APIs.

Q27. What is the difference between process.nextTick() and setImmediate()?

Answer:

1. **process.nextTick():**
    - Executes callbacks before the event loop continues.
    - Higher priority than setImmediate().
2. **setImmediate():**
    - Executes callbacks after the current poll phase.
3. **Use Case:**
    - Use process.nextTick() for immediate but short tasks.
    - Use setImmediate() for tasks that can wait until the event loop is idle.
4. **Order:**
    - process.nextTick() always runs before setImmediate() in the same iteration.
5. **Performance:**
    - Overusing process.nextTick() may block the event loop.

Event Emitters and Streams:

Q28. What are the different types of streams in Node.js?

Answer:

1. **Readable Streams:**
    - For reading data (e.g., fs.createReadStream).
    - Events: data, end, error.
2. **Writable Streams:**
    - For writing data (e.g., fs.createWriteStream).
    - Methods: write(), end().
3. **Duplex Streams:**
    - Both readable and writable (e.g., TCP sockets).
4. **Transform Streams:**
    - Modify or transform data while reading/writing (e.g., compression).
5. **Piping:**
    - Connect streams to transfer data seamlessly.

Q29. What are the key differences between Event Emitters and Streams?

Answer:

1. **Purpose:**
    - Event Emitters manage event-driven architecture.
    - Streams handle continuous data flows.
2. **Integration:**
    - Event Emitters are used to trigger and listen for custom events.
    - Streams use events for data flow management (data, end).
3. **Usage:**
    - Event Emitters are generic and can be used anywhere.
    - Streams are specifically for I/O operations.
4. **Examples:**
    - Event Emitter: Custom event handling.
    - Stream: File reading and writing.
5. **Built-In Use:**
    - Streams are based on EventEmitter.

Q30. Explain the pipe() method in Node.js streams.

Answer:
The pipe() method is used to connect readable streams to writable streams.

1. **Simplifies Data Flow:** Directly passes data from one stream to another.
2. **Chaining Streams:** Allows multiple streams to be connected in a chain.

```
const fs = require("fs");
const readStream = fs.createReadStream("input.txt");
const writeStream = fs.createWriteStream("output.txt");
readStream.pipe(writeStream);
```

3. **Event-Driven:** Automatically handles data and end events.
4. **Error Handling:** Use .on('error') for robust error handling

Q31. How does backpressure work in Node.js streams?

Answer:
Backpressure occurs when a writable stream can't handle data as fast as it's received.

1. **Data Overflow Prevention:** Regulates data flow between streams.
2. **Writable Stream Buffer:** Temporarily stores excess data.
3. **HighWaterMark:** Defines the buffer limit for writable streams.
4. **drain Event:** Signals that the writable stream is ready for more data.
5. **Example:**
    - Pause a readable stream until the writable stream is ready.

Q32. What is the relationship between Event Emitters and Streams?

Answer:

1. **Streams are Event Emitters:** All streams inherit from the EventEmitter class.
2. **Event-Driven Behavior:** Streams emit events like data, end, and error.
3. **Reusability:** Event Emitter logic powers the functionality of streams.
4. **Error Handling:** Use on('error') to manage stream errors.
5. **Custom Events:** Developers can extend stream classes to emit custom events.

Buffers and File Operations:

Q33. How do you create and manipulate Buffers in Node.js?

Answer:

1. **Creating a Buffer:**

```
const buf = Buffer.alloc(10); // Allocates 10 bytes
const bufFrom = Buffer.from("Hello"); // Creates from string
```

Writing Data to a Buffer:

```
buf.write("Node.js");
```

Reading Data from a Buffer:

```
console.log(buf.toString());
```

Buffer Length:

```
console.log(buf.length);
```

Modify Buffer Data:
Buffers can be modified directly using byte indices.

Q34. What is the difference between Buffer.alloc() and Buffer.allocUnsafe()?

Answer:

1. **Buffer.alloc(size):**
   - Allocates memory and initializes it to zero.
   - Safer but slower due to initialization overhead.
2. **Buffer.allocUnsafe(size):**
   - Allocates memory without initializing it.
   - Faster but may contain old data from memory.
3. **Use Case:**
   - Use allocUnsafe for performance-critical tasks where the buffer will be fully overwritten.
4. **Security:**
   - Avoid allocUnsafe in sensitive applications to prevent data leaks.

Q35. How do you convert a Buffer to JSON or Base64 in Node.js?

Answer:

1. Buffer to JSON:

```
const buf = Buffer.from("Hello");
console.log(buf.toJSON());
```

2. Buffer to Base64:

```
const base64 = buf.toString("base64");
console.log(base64);
```

3. Base64 to Buffer:

```
const decoded = Buffer.from(base64, "base64");
console.log(decoded.toString());
```

4. Use Cases:

JSON conversion for APIs.
Base64 for encoding binary data in strings.

Q36. What are some best practices for working with Buffers in Node.js?

Answer:

1. **Allocate Safely:** Use Buffer.alloc for secure memory allocation.
2. **Avoid Large Buffers:** Use streams to handle large data instead of large buffers.
3. **Encoding Management:** Always specify the correct encoding when converting.
4. **Error Handling:** Check for errors during file and buffer operations.
5. **Cleanup:** Release unused buffers to free up memory.

Child Processes and Worker Threads:

Q37.What are child processes in Node.js, and why are they used?

Answer:
Child processes are independent processes spawned by a Node.js application to perform parallel tasks.

1. **Parallel Execution:** Allows Node.js to run multiple operations simultaneously.
2. **CPU-Intensive Tasks:** Offloads heavy tasks, such as computations, to child processes.
3. **Methods for Creation:** Use spawn(), exec(), execFile(), or fork().
4. **Communication:** Parent and child processes communicate using IPC (Inter-Process Communication).
5. **Scalability:** Improves application scalability by leveraging multiple CPU cores.
6. **Built-In Module:** Requires the child_process module for implementation.

Q38. What are the differences between spawn(), exec(), execFile(), and fork() in the child_process module?

Answer:

1. **spawn()**
   - Launches a new process with a specified command.
   - Streams data for real-time interaction.
   - Example: Streaming large output of a script.
2. **exec()**
   - Executes a command and buffers the output (stdout and stderr).
   - Not suitable for large output due to buffer size limitation.
3. **execFile()**
   - Directly executes a file without spawning a shell.
   - Faster and more secure than exec().
4. **fork()**
   - Specifically for spawning Node.js scripts.
   - Provides built-in IPC for communication between processes.
5. **Use Cases:**
   - spawn: Streaming tasks.

- exec: Commands with small output.
- execFile: External scripts.
- fork: Node.js script communication.

Q39. What are Worker Threads in Node.js?

Answer:
Worker Threads are used to execute JavaScript code in parallel, using multiple threads within the same process.

1. **Thread-Based Parallelism:** Provides an alternative to process-based parallelism.
2. **CPU-Intensive Tasks:** Ideal for computational tasks that block the event loop.
3. **Shared Memory:** Allows threads to share memory via SharedArrayBuffer.
4. **Isolated Contexts:** Each thread runs in its own V8 engine instance.
5. **Communication:** Uses message passing for interaction between threads.
6. **Built-In Module:** Requires the worker_threads module.

Q40. What is the difference between child processes and worker threads?

Answer:

1. **Process vs. Thread:**
   - Child processes create separate processes.
   - Worker threads run within the same process.
2. **Memory Usage:**
   - Child processes consume more memory as they have independent instances.
   - Worker threads are more memory-efficient due to shared resources.
3. **Use Cases:**
   - Child processes: External scripts, tasks requiring isolation.
   - Worker threads: Computational tasks in the same application.
4. **Communication:**
   - Both use message passing, but worker threads can also share memory.
5. **Performance:**
   - Worker threads are faster for tasks involving shared memory.

Q41. Provide an example of using a worker thread in Node.js.

Answer:

Main Thread:

```
const { Worker } = require("worker_threads");

const worker = new Worker("./worker.js", { workerData: { num: 42 } });

worker.on("message", (result) => {
  console.log(`Result: ${result}`);
});

worker.on("error", (err) => {
```

```
  console.error(`Error: ${err}`);
});

worker.on("exit", (code) => {
  console.log(`Worker exited with code ${code}`);
});
```

Worker Thread (worker.js):

```
const { parentPort, workerData } = require("worker_threads");

const result = workerData.num * 2;
parentPort.postMessage(result);
```

Output:

- Result: 84

Q42. What are the key events in child processes?

Answer:

1. **exit:** Triggered when the process exits.
2. **close:** Emitted when all stdio streams are closed.
3. **error:** Fired when an error occurs during spawning or execution.
4. **disconnect:** Occurs when the IPC channel is closed.
5. **message:** Used for message passing between parent and child processes.

Q43. What are some best practices for using child processes and worker threads in Node.js?

Answer:

1. **Choose Wisely:** Use child processes for isolated tasks and worker threads for shared tasks.
2. **Limit Resource Usage:** Avoid spawning too many processes or threads.
3. **Error Handling:** Always handle errors in both parent and child/worker contexts.
4. **Monitor Performance:** Use tools to track CPU and memory usage.
5. **Graceful Shutdown:** Ensure processes and threads are terminated properly.
6. **Security:** Validate input data to prevent vulnerabilities.

Express.js

Introduction to Express.js:

Q44. What is Express.js, and why is it used?

Answer:
Express.js is a minimal and flexible Node.js web application framework used to build server-side applications.

1. **Simplifies Web Development:** Provides tools and utilities for handling HTTP requests and responses.
2. **Middleware Support:** Allows adding custom middleware for pre-processing requests.
3. **Routing:** Simplifies URL routing and request handling.
4. **Scalable Applications:** Suitable for building single-page, multi-page, and hybrid web applications.
5. **Extensible:** Supports plugins and third-party middleware for extended functionality.
6. **Fast Development:** Reduces boilerplate code, speeding up the development process.

Q45. How do you set up an Express.js application?

Answer:

1. Install Express:

```
npm install express
```

2. Basic Setup:

```
const express = require("express");
const app = express();

app.get("/", (req, res) => {
  res.send("Hello, Express!");
});

app.listen(3000, () => {
  console.log("Server is running on port 3000");
});
```

3. **Handle Requests:** Define routes using app.get(), app.post(), app.put(), and app.delete()
4. **Middleware:** Use middleware functions to process requests before sending a response.
5. **Start Server:** Use app.listen() to start the application

Q46. What is middleware in Express.js?

Answer:
Middleware functions are functions that execute during the request-response cycle in an Express.js application.

1. **Pre-Processing Requests:** Modify or handle requests before passing them to the next handler.
2. **Access to Request and Response Objects:** Can read or modify req and res.

3. **Types of Middleware:** Built-in (express.json()), third-party (cors), and custom middleware.
4. **Chaining:** Middleware functions can be chained using next().
5. **Error Handling Middleware:** Specifically designed to handle errors in the app.

```
app.use((req, res, next) => {
  console.log("Middleware executed");
  next();
});
```

Q47. What is the role of routing in Express.js?

Answer:
Routing in Express.js defines how an application responds to client requests for specific URLs and HTTP methods.

1. **HTTP Method Handling:** Supports GET, POST, PUT, DELETE, etc.
2. **Dynamic Parameters:** Routes can include dynamic parameters using :.
3. **Middleware Integration:** Routes can use middleware for pre-processing.
4. **Route Grouping:** Organize routes using express.Router().

```
app.get("/user/:id", (req, res) => {
  res.send(`User ID: ${req.params.id}`);
});
```

Q48. What are some built-in middleware functions in Express.js?

Answer:

1. **express.json():** Parses incoming JSON requests and puts the data in req.body.
2. **express.urlencoded():** Parses URL-encoded data (form submissions).
3. **express.static():** Serves static files like images, CSS, and JavaScript.
4. **express.text():** Parses plain text requests.
5. **Error Handling Middleware:** Captures and processes errors in the app.

Q49. What is the difference between app.use() and route-specific methods (app.get(), app.post())?

Answer:

app.use():

- Used to apply middleware to all routes or specific route patterns.

```
app.use((req, res, next) => {
  console.log("Middleware for all routes");
  next();
});
```

Route-Specific Methods:

- Handle specific HTTP methods for particular paths.

```
app.get("/home", (req, res) => res.send("GET /home"));
```

Global vs. Specific:

- app.use() is for middleware, route methods define responses.

Q50. What are some common use cases for Express.js?

Answer:

1. **RESTful APIs:** Create APIs with simple route handling.
2. **Static File Hosting:** Serve HTML, CSS, and JavaScript files.
3. **Web Applications:** Build single-page and multi-page web apps.
4. **Middleware Integration:** Use for logging, authentication, or error handling.
5. **Real-Time Applications:** Integrates well with tools like WebSockets for chat apps.
6. **Microservices:** Lightweight framework for developing microservices.

Middleware in Express.js:

Q51. What is middleware in Express.js?

Answer:
Middleware in Express.js is a function executed during the request-response cycle that processes the incoming request before sending the response.

1. **Request Pre-Processing:** Modifies or processes the request object.
2. **Response Handling:** Can end the response or pass control to the next middleware.
3. **Types:** Built-in, third-party, and custom middleware.
4. **next() Function:** Middleware uses next() to move to the next function in the chain.
5. **Global or Route-Specific:** Can be applied globally or to specific routes.
6. **Error Handling:** Special middleware handles errors in the application.

Q52. What are the types of middleware in Express.js?

Answer:

1. **Built-In Middleware:**
   - Example: express.json() for parsing JSON requests.
   - Comes pre-installed with Express.js.
2. **Third-Party Middleware:**
   - Example: cors, morgan, body-parser.
   - Installable via NPM for extended functionality.
3. **Custom Middleware:**
   - Created by developers for specific tasks like logging or authentication.
4. **Error-Handling Middleware:**
   - Handles errors using four arguments: (err, req, res, next).

5. **Static Middleware:**
    - Example: express.static() for serving static files.

Q53. How do you create custom middleware in Express.js?

Answer:

1. **Define Middleware:**

```
const logger = (req, res, next) => {
  console.log(`Request URL: ${req.url}`);
  next(); // Pass control to the next middleware
};
```

2. Apply Globally:

```
app.use(logger);
```

3. Apply to Specific Route:

```
app.get("/user", logger, (req, res) => {
  res.send("User Page");
});
```

4. **Execution:** Middleware runs before the route handler processes the request
5. **Chaining:** Use next() to continue to the next middleware or route handler.

Q54. What is the role of the next() function in middleware?

Answer:

1. **Control Flow:** Passes control to the next middleware or route handler.
2. **Asynchronous Handling:** Ensures asynchronous tasks complete before moving to the next function.
3. **Error Handling:** Call next(err) to pass errors to error-handling middleware.
4. **Skips Middleware:** Bypasses the current middleware if next() is called.
5. **Mandatory Call:** Failure to call next() can result in a stalled request.

```
app.use((req, res, next) => {
  console.log("Middleware executed");
  next();
});
```

Q55. What are some commonly used third-party middleware in Express.js?

Answer:

1. **cors:** Handles Cross-Origin Resource Sharing (CORS).
2. **morgan:** Logs HTTP requests for debugging.

3. **body-parser:** Parses incoming request bodies (now integrated as express.json()).
4. **cookie-parser:** Parses cookies attached to client requests.
5. **helmet:** Provides security headers to protect against web vulnerabilities.
6. **compression:** Compresses response bodies for better performance.

Q56. What is built-in middleware in Express.js?

Answer:

1. **express.json():** Parses JSON request bodies.
2. **express.urlencoded():** Parses URL-encoded data from forms.
3. **express.static():** Serves static files like CSS, JS, and images.
4. **express.text():** Parses plain text request bodies.
5. **Purpose:** Built-in middleware simplifies handling common tasks without external dependencies.

```
app.use(express.json());
app.use(express.static("public"));
```

Q57. What is the difference between application-level and route-level middleware?

Answer:

Application-Level Middleware:

- Applied globally to all routes.

```
app.use(logger);
```

Route-Level Middleware:

- Applied to specific routes.

```
app.get("/user", authMiddleware, (req, res) => res.send("User Page"));
```

Scope: Application-level middleware affects all routes; route-level affects only specific routes.

Use Cases:

- Application-level: Logging, parsing.
- Route-level: Authentication, validation.

Advanced Routing:

Q58. What is routing in Express.js, and how does advanced routing differ from basic routing?

Answer:
Routing in Express.js defines how an application responds to client requests for specific URLs and HTTP methods.

1. **Basic Routing:** Handles simple routes with methods like app.get() or app.post().
2. **Advanced Routing:** Includes dynamic routes, route grouping, route parameters, and middleware chaining.
3. **Dynamic Parameters:** Allows routes to accept variable data with : syntax.
4. **Route Grouping:** Organizes routes logically using express.Router().
5. **Chaining Handlers:** Supports multiple handlers for a single route.
6. **Use Cases:** Builds complex APIs with reusable and modular routing logic.

Q59. What are route parameters, and how are they used in Express.js?

Answer:

1. **Definition:** Dynamic parts of a route path specified with :.
2. **Access Parameters:** Use req.params to retrieve parameter values.

```
app.get("/user/:id", (req, res) => {
  res.send(`User ID: ${req.params.id}`);
});
```

3. **Multiple Parameters:** Define routes with multiple dynamic segments.

```
app.get("/product/:category/:id", (req, res) => {
  res.send(req.params);
});
```

4. **Validation:** Middleware can validate parameters before processing the request.
5. **Error Handling:** Ensure proper error handling for missing or invalid parameters.

Q60. What is route grouping in Express.js, and how is it implemented?

Answer:

1. Purpose: Organizes related routes into a single module using express.Router().
2. Create a Router:

```
const express = require("express");
const router = express.Router();

router.get("/profile", (req, res) => res.send("Profile Page"));
router.get("/settings", (req, res) => res.send("Settings Page"));
module.exports = router;
```

3. Integrate Router:

```
const userRoutes = require("./routes/user");
```

```
app.use("/user", userRoutes);
```

4. Modularity: Improves code readability and maintainability.
5. Middleware Integration: Apply middleware to specific route groups.

Q61. How do you handle multiple route handlers for the same path?

Answer:

1. Chaining Handlers: Use an array or chain of middleware functions for a route.

```
app.get(
  "/example",
  (req, res, next) => {
    console.log("First handler");
    next();
  },
  (req, res) => {
    res.send("Second handler");
  }
);
```

2. Middleware for Preprocessing: Process data or validate requests before reaching the final handler.
3. Error Handling: Insert error-handling middleware in the chain if necessary.
4. Example Use Case: Authentication, logging, or input validation.

Q62. How does Express.js handle route precedence and conflicts?

Answer:

1. Order Matters**:** Routes are matched in the order they are defined.
2. Specificity: Define more specific routes before generic ones.
   Example: Place /user/:id before /user.
3. Wildcard Routes: Use cautiously as they may override other routes.

```
app.get("*", (req, res) => res.send("Catch-all route"));
```

4. Route Conflicts**:** Avoid overlapping routes that cause ambiguity.
5. Middleware Precedence**:** Ensure middleware doesn't interfere with route logic.

Q63. What is route chaining, and how is it implemented in Express.js?

Answer:

1. Definition**:** Attaching multiple HTTP methods to the same route path.
2. Implementation**:** Use app.route().

```
app
  .route("/user")
  .get((req, res) => res.send("Get User"))
  .post((req, res) => res.send("Create User"))
  .put((req, res) => res.send("Update User"));
```

3. **Reusability:** Reduces redundancy by grouping handlers for the same route
4. **Middleware Integration:** Apply middleware to specific methods.
5. **Readability:** Improves code organization.

Templating and Views:

Q64. What is templating in Express.js, and why is it important?

Answer:
Templating in Express.js involves using template engines to dynamically render HTML pages with data.

1. **Dynamic Content:** Generates HTML pages with dynamic data from the server.
2. **Code Reusability:** Enables reusing templates for consistent layouts across multiple pages.
3. **Separation of Concerns:** Separates presentation logic from application logic.
4. **Template Engines:** Supports engines like EJS, Pug, and Handlebars for rendering views.
5. **Faster Development:** Simplifies rendering dynamic data into pre-designed HTML structures.
6. **Interactive Pages:** Enables creating interactive and user-specific content.

Q65. What is a view engine, and how does it work in Express.js?

Answer:

1. **Definition:** A view engine is a tool that processes templates and converts them into HTML.
2. **Integration with Express:** Set the view engine in Express using the app.set() method.

```
app.set("view engine", "ejs");
```

3. **Render Views:** Use res.render() to render templates.

```
res.render("index", { title: "Home Page" });
```

4. **Supported Engines:** Popular ones include EJS, Pug (formerly Jade), and Handlebars.
5. **Data Binding:** Pass dynamic data from the server to the template for rendering.
6. **Directory Configuration:** Define the views directory using app.set('views', path).

Node.js, Express.js and MongoDB 150+ Theory Interview Questions

Q66. What is the difference between server-side rendering (SSR) and client-side rendering (CSR)?

Answer:

1. **Server-Side Rendering (SSR):**
   - HTML is generated on the server and sent to the client.
   - Example: Express.js with a templating engine like Pug or EJS.
   - Pros: Faster initial load, better SEO.
   - Cons: Increased server load.
2. **Client-Side Rendering (CSR):**
   - HTML is rendered on the client using JavaScript frameworks like React.
   - Pros: Better interactivity, reduced server load.
   - Cons: Slower initial load, relies on JavaScript.
3. **Combination:** Modern applications use both SSR and CSR for optimal performance.

Q67. How does res.render() work in Express.js?

Answer:

1. **Render Templates:** Converts a template into HTML using the view engine.

```
res.render("template", { key: value });
```

2. **Pass Data:** Dynamically bind data to the template using an object.
3. **Default Directory:** Searches templates in the directory set by app.set('views').
4. **Error Handling:** Throws an error if the template is not found or invalid.
5. **Integration with Middleware:** Works seamlessly with middleware to pass pre-processed data.

Q68. What are some popular templating engines supported by Express.js?

Answer:

1. **EJS (Embedded JavaScript):**
   - Syntax similar to HTML with JavaScript embedding.
2. **Pug (formerly Jade):**
   - Minimal syntax with indentation-based structure.
3. **Handlebars:**
   - Extends Mustache with helpers and custom logic.
4. **Mustache:**
   - Simple and logic-less templating.
5. **Nunjucks:**
   - Flexible engine with extensive features.
6. **Use Case:** Choose based on project requirements and team familiarity.

RESTful APIs with Express.js:

Node.js, Express.js and MongoDB 150+ Theory Interview Questions

Q69. What is a RESTful API, and why is it used?

Answer:
A RESTful API (Representational State Transfer API) is a standardized approach for building APIs that use HTTP methods to interact with resources.

1. **Stateless Communication:** Each request contains all the necessary information, making the server stateless.
2. **Resource-Based:** Uses URLs to represent resources, like /users or /products.
3. **HTTP Methods:** Employs methods like GET, POST, PUT, DELETE for CRUD operations.
4. **Scalability:** Enables scalable and modular design for client-server communication.
5. **Language Agnostic:** Works with any language or platform supporting HTTP.
6. **Lightweight:** Uses simple data formats like JSON or XML for data exchange.

Q70. How do you set up a basic RESTful API in Express.js?

Answer:

1. Install Express:

```
npm install express
```

2. Basic Setup:

```
const express = require("express");
const app = express();
app.use(express.json());

app.get("/api/users", (req, res) => res.send("GET Users"));
app.post("/api/users", (req, res) => res.send("POST User"));
app.put("/api/users/:id", (req, res) => res.send(`PUT User ${req.params.id}`));
app.delete("/api/users/:id", (req, res) =>
  res.send(`DELETE User ${req.params.id}`)
);

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

3. **Route Definition:** Use HTTP methods for CRUD operations.
4. **Middleware:** Parse request bodies and handle errors.
5. **Data Handling:** Integrate with a database for real-world use.

Q71. What are the key HTTP methods used in RESTful APIs?

Answer:

1. **GET:**
   - Retrieve data from the server.
   - Example: Fetch user details with /users/123.

2. **POST:**
    - Create a new resource.
    - Example: Add a new user with /users.
3. **PUT:**
    - Update an existing resource.
    - Example: Update user data with /users/123.
4. **DELETE:**
    - Remove a resource.
    - Example: Delete a user with /users/123.
5. **PATCH:**
    - Partially update a resource.
    - Example: Update the email of a user with /users/123.

Q72. What is req.params, and how is it used in RESTful APIs?

Answer:

1. **Definition:** Captures route parameters from dynamic URLs.
2. **Access Parameters:** Use req.params to retrieve values.

```
app.get("/users/:id", (req, res) => {
  res.send(`User ID: ${req.params.id}`);
});
```

3. **Dynamic Routing:** Allows flexibility in defining resource-specific routes.
4. **Validation:** Validate req.params to prevent incorrect or malicious data.
5. **Example Use Case:** Fetch details for a specific user or product.

Q73. What is the purpose of middleware in RESTful APIs?

Answer:

1. **Request Parsing:** Use middleware like express.json() to parse JSON request bodies.
2. **Authentication:** Verify user identity for protected routes.
3. **Logging:** Log API requests using middleware like morgan.
4. **Error Handling:** Centralize error responses for consistency.
5. **Cross-Origin Requests:** Handle CORS issues using cors middleware.

```
const cors = require("cors");
app.use(cors());
```

Q74. How do you handle errors in RESTful APIs?

Answer:

1. **Error-Handling Middleware:**

```
app.use((err, req, res, next) => {
  res.status(500).json({ error: err.message });
});
```

2. **Consistent Responses:** Standardize error structures across endpoints.
3. **HTTP Status Codes:** Use appropriate status codes like 400 (Bad Request) or 500 (Internal Server Error).
4. **Validation Errors:** Return meaningful messages for invalid input.
5. **Logging:** Log errors for debugging and monitoring purposes.

Authentication and Authorization:

Q75. What is the difference between authentication and authorization?

Answer:
Authentication and authorization are security processes with distinct purposes.

1. **Authentication:**
   - Verifies the identity of a user.
   - Example: Logging in with a username and password.
2. **Authorization:**
   - Determines what resources a user has access to.
   - Example: Allowing an admin to delete users but restricting regular users.
3. **Sequence:** Authentication occurs before authorization.
4. **Scope:** Authentication validates "who you are," while authorization defines "what you can do."
5. **Use Case:** Both processes are essential for secure applications.

Q76. What are common methods of authentication in web applications?

Answer:

1. **Password-Based Authentication:**
   - Users log in with a username and password.
   - Example: Traditional login forms.
2. **Token-Based Authentication:**
   - Uses tokens like JWT (JSON Web Tokens) to validate users.
3. **OAuth:**
   - Allows third-party authentication (e.g., "Log in with Google").
4. **Session-Based Authentication:**
   - Stores session data on the server for logged-in users.
5. **Multi-Factor Authentication (MFA):**
   - Combines two or more methods, such as passwords and OTPs.
6. **Biometric Authentication:**
   - Uses fingerprints, facial recognition, or retina scans.

Q77. How does token-based authentication work?

Answer:

1. **Login Request:** User sends credentials to the server.
2. **Token Issuance:** Server verifies credentials and returns a token (e.g., JWT).

3. **Client Storage:** Token is stored on the client-side (e.g., local storage or cookies).
4. **Subsequent Requests:** Client sends the token in the Authorization header.
5. **Validation:** Server validates the token to grant or deny access.
6. **Stateless Design:** Server doesn't need to store session data, making it scalable.

Q78. What is JWT (JSON Web Token), and how is it used?

Answer:
JWT is a compact, self-contained token used for secure communication between parties.

1. **Structure:** Contains three parts: Header, Payload, and Signature.
2. **Stateless:** Encodes all necessary information within the token, removing the need for server-side storage.
3. **Use Case:** Authentication and secure information exchange.
4. **Validation:** Verified using a secret key or public/private key pair.
5. **Example:**
    - Authorization: Bearer <token>.
6. **Expiration:** Tokens include expiration times for added security.

Q79. What is the difference between session-based and token-based authentication?

Answer:

1. **Session-Based Authentication:**
    - Stores user sessions on the server.
    - Example: PHP sessions or server cookies.
    - Requires server-side state management.
2. **Token-Based Authentication:**
    - Uses tokens like JWT for stateless communication.
    - Example: SPA authentication.
3. **Scalability:**
    - Token-based authentication is more scalable due to its stateless nature.
4. **Security:**
    - Tokens are vulnerable to client-side storage attacks if not handled securely.
5. **Use Cases:**
    - Session-based: Traditional web apps.
    - Token-based: APIs and microservices.

Q80. What is OAuth, and how does it enable third-party authentication?

Answer:

1. **Definition:** OAuth (Open Authorization) is a protocol for third-party access without sharing passwords.
2. **Example:** "Log in with Google" or "Log in with Facebook."
3. **Roles in OAuth:**
    - **Resource Owner:** User who owns the data.
    - **Client:** Application requesting access.
    - **Authorization Server:** Issues access tokens.
    - **Resource Server:** API that hosts the user's data.

4. **Authorization Code Flow:** Secure method for exchanging access tokens.
5. **Security:** Reduces risk by not sharing user credentials with third parties.

Q81. What is CORS, and why is it important in authentication?

Answer:
CORS (Cross-Origin Resource Sharing) is a security feature that controls how resources are shared across domains.

1. **Same-Origin Policy:** Restricts requests from different origins for security reasons.
2. **CORS Headers:** Use headers like Access-Control-Allow-Origin to specify allowed origins.
3. **Authentication Scenarios:** Required when making API requests from a front-end hosted on a different domain.
4. **Middleware:** Use the cors package in Express.js.

```
const cors = require("cors");
app.use(cors({ origin: "https://example.com" }));
```

5. **Security:** Ensures safe cross-origin communication.

File Uploads and Static Assets:

Q82. How does Express.js handle file uploads?

Answer:
Express.js handles file uploads using middleware like multer.

1. **Middleware for Parsing:** Use multer to parse multipart/form-data, the format for file uploads.
2. **Storage Options:** Supports memory storage (temporary) or disk storage (permanent).
3. **Single vs. Multiple Files:** Allows uploading single or multiple files.

```
const upload = multer({ dest: "uploads/" });
app.post("/upload", upload.single("file"), (req, res) => {
  res.send("File uploaded successfully");
});
```

4. **Names:** Specify field names for uploaded files.
5. **Validation:** Validate file types, sizes, and other properties before saving.

Q83. What is multer, and how do you use it for file uploads?

Answer:
multer is a Node.js middleware for handling multipart/form-data.

1. Install Multer:

```
npm install multer
```

2. Setup Multer:

```
const multer = require("multer");
const upload = multer({ dest: "uploads/" });
```

3. Single File Upload:

```
app.post("/upload", upload.single("file"), (req, res) => {
  res.send(`File uploaded: ${req.file.filename}`);
});
```

4. Multiple File Uploads:

```
app.post("/uploads", upload.array("files", 5), (req, res) => {
  res.send("Multiple files uploaded");
});
```

5. **Validation:** Use filters to check file types and sizes.

Q84. What are some best practices for handling file uploads in Express.js?

Answer:

1. **Limit File Sizes:** Prevent server overload by limiting file sizes.
2. **Validate File Types:** Accept only allowed file formats for security.
3. **Store Securely:** Save files in restricted directories or use cloud storage services.
4. **Sanitize File Names:** Remove or escape special characters to prevent malicious input.
5. **Use HTTPS:** Encrypt file uploads to prevent data interception.
6. **Scan for Malware:** Use antivirus software to scan uploaded files.

Q85. What are the differences between memory storage and disk storage in multer?

Answer:

1. **Memory Storage:**
   - Stores files in memory as Buffer objects.
   - Suitable for processing files directly in code.
   - Example: Image manipulation.
2. **Disk Storage:**
   - Saves files to a specified directory on disk.
   - Suitable for long-term storage or transferring files to external services.
   - Example: File archiving.
3. **Configuration Example:**

```
const storage = multer.diskStorage({
  destination: (req, file, cb) => cb(null, "uploads/"),
  filename: (req, file, cb) => cb(null, Date.now() + "-" + file.originalname),
```

```
});
const upload = multer({ storage });
```

4. **Security:** Memory storage minimizes disk access but increases RAM usage.

Q86. What are some considerations for serving static assets in production?

Answer:

1. **CDN:** Use a Content Delivery Network (CDN) for faster delivery of assets.
2. **Caching:** Enable caching with proper HTTP headers.
3. **Compression:** Compress files using Gzip or Brotli for reduced load times.
4. **Minification:** Minify CSS, JavaScript, and HTML files to improve performance.
5. **Security Headers:** Use middleware like helmet to set secure HTTP headers.
6. **Versioning:** Add version numbers to assets for cache invalidation during updates.

Q87. How do you combine file uploads with API endpoints?

Answer:

1. **Middleware Integration:** Use multer as middleware for specific routes.
2. **Route Example:**

```
app.post("/api/upload", upload.single("file"), (req, res) => {
  res.json({ file: req.file });
});
```

3. **Return Metadata:** Include file metadata in the API response.
4. **Database Storage:** Save file references (e.g., file name, path) in the database.
5. **Error Handling:** Ensure the API returns proper status codes and messages for errors.

Introduction to MongoDB:

Q88. What is MongoDB, and why is it widely used?

Answer:
MongoDB is a NoSQL, document-oriented database that stores data in flexible, JSON-like documents.

1. **NoSQL Database:** Unlike relational databases, it doesn't rely on tables and rows but uses collections and documents.
2. **Schema Flexibility:** Allows dynamic, schema-less data models, making it highly adaptable.
3. **Document-Oriented:** Stores data in BSON (binary JSON) format for easy readability and flexibility.
4. **Scalability:** Supports horizontal scaling through sharding for handling large data sets.
5. **Cross-Platform:** Runs on multiple operating systems like Windows, Linux, and macOS.
6. **Rich Query Language:** Offers powerful querying, aggregation, and indexing features.

Node.js, Express.js and MongoDB 150+ Theory Interview Questions

Q89. How is MongoDB different from relational databases?

Answer:

1. **Data Storage:**
   - MongoDB: Stores data in JSON-like documents.
   - RDBMS: Stores data in tables with rows and columns.
2. **Schema:**
   - MongoDB: Schema-less, allowing flexible data structures.
   - RDBMS: Requires a predefined schema.
3. **Scaling:**
   - MongoDB: Horizontally scalable through sharding.
   - RDBMS: Primarily scales vertically.
4. **Relationships:**
   - MongoDB: Uses embedded documents or references.
   - RDBMS: Relies on foreign keys for relationships.
5. **Query Language:**
   - MongoDB: Uses BSON-based queries.
   - RDBMS: Uses SQL.

Q90. What are collections and documents in MongoDB?

Answer:

1. **Collections:**
   - A group of documents, similar to tables in RDBMS.
   - Example: A users collection can store user data.
2. **Documents:**
   - The fundamental unit of data in MongoDB, stored as BSON.

```
{
  "_id": "12345",
  "name": "Mahesh",
  "email": "Mahesh@example.com"
}
```

3. **Schema Flexibility:** Documents within a collection can have different fields.
4. **Indexing:** Collections can have indexes for efficient querying.
5. **Dynamic Growth:** Collections grow automatically as documents are added.

Q91. What is BSON, and how does it differ from JSON?

Answer:

1. **Definition:** BSON (Binary JSON) is a binary-encoded format used by MongoDB to store documents.
2. **Performance:** Faster parsing and smaller storage size compared to JSON.
3. **Data Types:** Supports additional data types like Date, Binary, and ObjectId.
4. **Compactness:** Designed for efficient storage and traversal in databases.
5. **Example:** A BSON document looks similar to JSON but includes type metadata.

Node.js, Express.js and MongoDB 150+ Theory Interview Questions

Q92. What is sharding in MongoDB?

Answer:

1. **Definition:** Sharding is a method of distributing data across multiple servers for horizontal scaling.
2. **Shard Keys:** Data is divided based on a shard key.
3. **Improves Performance:** Handles large datasets by spreading the load across servers.
4. **High Availability:** Ensures data redundancy and reliability.
5. **Use Case:** Suitable for applications with high write/read operations.
6. **MongoDB Configuration:** Managed through a config server and shards.

Q93. What is a replica set in MongoDB?

Answer:

1. **Definition:** A group of MongoDB servers that maintain the same data for redundancy and high availability.
2. **Primary Node:** Handles read and write operations.
3. **Secondary Nodes:** Replicates data from the primary and can serve read requests.
4. **Automatic Failover:** Elects a new primary in case of failure.
5. **Use Case:** Ensures data availability during server downtime.
6. **Configuration:** Defined during MongoDB setup with multiple instances.

Q94. How does MongoDB handle indexing?

Answer:

1. **Purpose:** Speeds up query execution by allowing efficient data retrieval.
2. **Default _id Index:** Every collection has a default index on the _id field.
3. **Custom Indexes:** Create indexes on specific fields to optimize queries.

```
db.collection.createIndex({ name: 1 }); // Ascending index
```

4. **Compound Indexes:** Supports indexing on multiple fields.
5. **Text Indexes:** Allows searching text-based data with text indexes.
6. **TTL Indexes:** Automatically removes documents after a specified time.

Q95. What is the aggregation framework in MongoDB?

Answer:

1. **Purpose:** Performs advanced data processing and transformation on collections.
2. **Pipeline Stages:** Processes data through stages like $match, $group, $sort, $project.

```
db.collection.aggregate([
  { $match: { status: "active" } },
  { $group: { _id: "$category", total: { $sum: 1 } } },
]);
```

3. **Real-Time Analytics:** Useful for generating reports and insights.
4. **Optimization:** Executes queries efficiently using indexes.

MongoDB Data Modeling:

Q96. What is data modeling in MongoDB, and why is it important?

Answer:
Data modeling in MongoDB involves designing the structure of documents and collections to optimize storage and performance.

1. **Schema Design:** Defines how data is organized within documents and collections.
2. **Optimized Query Performance:** A well-structured model improves read and write efficiency.
3. **Scalability:** Supports large-scale data distribution with sharding and indexing.
4. **Application Requirements:** Aligns the data model with application-specific use cases.
5. **Schema Flexibility:** MongoDB allows dynamic and schema-less designs for evolving requirements.
6. **Minimizes Redundancy:** Reduces duplicate data through proper structuring

Q97. What are the key principles of data modeling in MongoDB?

Answer:

1. **Understand Application Workloads:** Focus on query patterns and use cases before designing.
2. **Embed vs. Reference:** Decide between embedding data or using references based on relationships.
3. **Data Access Patterns:** Optimize for frequently accessed data to minimize expensive operations.
4. **Denormalization:** Duplicate data strategically to improve read performance.
5. **Indexing:** Create indexes to support common queries.
6. **Scalability Considerations:** Ensure the model supports horizontal scaling if required.

Q98. What is the difference between embedding and referencing in MongoDB?

Answer:

1. **Embedding:**
    - Stores related data within a single document.
    - Example:

```
{
  "name": "Mahesh",
  "orders": [
    { "product": "Laptop", "price": 1200 },
    { "product": "Phone", "price": 800 }
  ]
}
```

- Pros: Reduces the need for joins and improves query performance.
- Cons: Leads to large documents for high-volume relationships.

2. **Referencing:**
   - Stores related data in separate collections with references.
   - Example:

```
{ "name": "Mahesh", "orders": [ObjectId("orderId1"), ObjectId("orderId2")] }
```

   - Pros: Maintains data normalization and smaller document sizes.
   - Cons: Increases query complexity.
3. **Use Cases:**
   - Embed for one-to-few relationships.
   - Reference for one-to-many or many-to-many relationships.

Q99. What are MongoDB relationships, and how are they modeled?

Answer:

1. **One-to-One:**
   - Example: User profile stored in the same or separate collections.
   - Use embedding for faster access or referencing for modularity.
2. **One-to-Many:**
   - Example: A blog post with comments.
   - Use embedding for few related items, referencing for large datasets.
3. **Many-to-Many:**
   - Example: Students enrolled in courses.
   - Use referencing with a linking collection.
4. **Denormalization:** Duplicate data for improved read performance where necessary.
5. **Trade-Offs:** Choose the modeling strategy based on query needs and data growth.

Q100. What are the best practices for designing data models in MongoDB?

Answer:

1. **Analyze Queries:** Model data based on frequent query patterns.
2. **Embed Data for Efficiency:** Embed related data if access patterns favor a single read.
3. **Use References for Scalability:** Use references when relationships require modularity or large datasets.
4. **Index Strategically:** Create indexes for fields used in queries and sorting.
5. **Limit Document Size:** Keep document sizes under 16MB for better performance.
6. **Avoid Over-Nesting:** Limit nesting depth to prevent complex queries.

Q101. How do you optimize MongoDB data models for aggregation?

Answer:

1. **Pre-Aggregation:** Store aggregated results in a separate collection for frequent use.
2. **Denormalization:** Include pre-computed fields to reduce aggregation complexity.

3. **Use $lookup:** Use the aggregation framework's $lookup for combining data from multiple collections.
4. **Pipeline Efficiency:** Optimize pipeline stages like $match, $group, and $sort for minimal resource use.
5. **Index Support:** Ensure indexes exist for fields used in the aggregation pipeline.

Q102. What is the trade-off between normalization and denormalization in MongoDB?

Answer:

1. **Normalization:**
   - Stores data in multiple collections to avoid redundancy.
   - Reduces storage usage but increases query complexity.
   - Example: Using references for relational data.
2. **Denormalization:**
   - Stores related data in the same document for faster access.
   - Increases storage usage but improves read performance.
   - Example: Embedding related fields directly.
3. **Use Case Considerations:**
   - Normalize for write-heavy applications.
   - Denormalize for read-heavy applications.

Working with Mongoose:

Q102. What is Mongoose, and why is it used?

Answer:
Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js.

1. **Simplified Database Interaction:** Provides a schema-based solution to model MongoDB data.
2. **Schema Definition:** Allows defining schemas with validation, default values, and constraints.
3. **Built-In Query Functions:** Offers methods for CRUD operations and advanced queries.
4. **Middleware Support:** Includes pre and post hooks for handling complex logic during queries.
5. **Data Validation:** Ensures that data adheres to the defined schema.
6. **Population:** Simplifies the handling of relationships between collections.

Q103. How do you set up and connect Mongoose to a MongoDB database?

Answer:

1. **Install Mongoose:**

```
npm install mongoose
```

2. **Connect to MongoDB:**

```
const mongoose = require("mongoose");
mongoose
  .connect("mongodb://localhost:27017/mydatabase", {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  })
  .then(() => console.log("Connected to MongoDB"))
  .catch((err) => console.error("Connection failed:", err));
```

3. **Connection Options:** Use useNewUrlParser and useUnifiedTopology for compatibility.
4. **Error Handling:** Handle connection errors using .catch() or on('error').
5. **Best Practices:** Store connection strings securely using environment variables.

Q104. What is a schema in Mongoose, and how is it created?

Answer:

1. **Definition:** A schema defines the structure of documents in a MongoDB collection.
2. **Creating a Schema:**

```
const mongoose = require("mongoose");
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: { type: Number, min: 0 },
});
```

3. **Field Validation:** Supports required fields, default values, and constraints like min or max.
4. **Timestamps:** Add createdAt and updatedAt fields using { timestamps: true }.
5. **Custom Methods:** Add custom methods or virtuals for advanced functionality.

Q105. What are Mongoose models, and how do they work?

Answer:

1. **Definition:** A model is a compiled version of a schema that interacts with the database.
2. **Creating a Model:**

```
const User = mongoose.model("User", userSchema);
```

3. **CRUD Operations:** Use models for creating, reading, updating, and deleting documents.

```
User.find({ name: "Alice" }).then((users) => console.log(users));
```

4. **Instance Methods:** Add methods directly to the schema for document-specific functionality.
5. **Static Methods:** Add methods to the model itself for collection-wide operations.

Q106. How does Mongoose handle data validation?

Answer:

1. **Built-In Validators:** Define constraints like required, unique, min, max, or match.

```
const userSchema = new mongoose.Schema({
  age: { type: Number, min: 0, max: 100 },
});
```

2. **Custom Validators:** Add custom validation logic.

```
email: {
  type: String,
  validate: {
    validator: (v) => /\S+@\S+\.\S+/.test(v),
    message: 'Invalid email format',
  },
}
```

3. **Asynchronous Validation:** Use async functions for complex validations.
4. **Middleware:** Combine validation with pre-save hooks for additional checks.
5. **Error Handling:** Capture validation errors in the catch block.

Q107. What is the difference between find(), findOne(), and findById() in Mongoose?

Answer:

1. **find():**
   - Returns all documents matching the query.
   - Example: User.find({ age: { $gt: 18 } }).
2. **findOne():**
   - Returns the first document matching the query.
   - Example: User.findOne({ email: 'test@example.com' }).
3. **findById():**
   - Finds a document by its _id field.
   - Example: User.findById('60c72b2f9f1b2c001f8e4abc').
4. **Performance:** findById() is optimized for queries on the _id field.
5. **Use Case:** Choose based on the need for multiple documents or a single record.

Relationships and Population in MongoDB:

Q108. What are relationships in MongoDB, and how are they modeled?

Answer:
Relationships in MongoDB represent the connection between documents in different collections or within the same collection.

1. **One-to-One:** A single document in one collection relates to a single document in another.
    - Example: A user profile linked to a single user.
2. **One-to-Many:** A single document relates to multiple documents in another collection.
    - Example: A blog post with multiple comments.
3. **Many-to-Many:** Multiple documents in one collection relate to multiple documents in another.
    - Example: Students enrolled in multiple courses.
4. **Embed or Reference:** Use embedding for closely related data, referencing for independent data.
5. **Flexibility:** MongoDB offers dynamic schema designs to accommodate changing relationships.

Q109. What is the difference between embedding and referencing in MongoDB relationships?

Answer:

1. **Embedding:**
    - Stores related data within the same document.
    - Example:

```
{
  "user": "Mahesh",
  "orders": [
    { "product": "Laptop", "quantity": 1 },
    { "product": "Mouse", "quantity": 2 }
  ]
}
```

    - Pros: Fast queries with fewer reads.
    - Cons: Increases document size, limited by 16MB document limit.
2. **Referencing:**
    - Stores related data in separate collections with references.
    - Example:

```
{ "user": "Mahesh", "orders": [ObjectId("orderId1"), ObjectId("orderId2")] }
```

    - Pros: Supports large datasets, smaller document size.
    - Cons: Requires additional queries or joins.
3. **Use Case:**
    - Embed for tightly coupled data.
    - Reference for loosely coupled or large data.

Q110. What is population in MongoDB, and how does it work in Mongoose?

Answer:

1. **Definition:** Population replaces a reference (ObjectId) in one collection with the actual document from another collection.
2. **Example:**

```
Post.find()
  .populate("author")
  .exec((err, posts) => {
    console.log(posts);
  });
```

3. **Usage:** Helps fetch related data without manually querying multiple collections.
4. **Nested Population:** Supports populating fields within populated documents.
5. **Performance Consideration:** Overuse can lead to slower queries.

Q111. How do you implement a one-to-one relationship in MongoDB?

Answer:

1. **Embedding:** Store the related document directly within the parent document.
    - Example:

```
{
  "userId": "123",
  "profile": { "age": 25, "location": "NYC" }
}
```

2. **Referencing:** Use ObjectId to reference another document.
    - Example:

```
{
  "userId": "123",
  "profileId": ObjectId("profile123")
}
```

3. **When to Use:**
    - Embed for static, small data.
    - Reference for large, frequently changing data.
4. **Indexing:** Add indexes for faster lookups.
5. **Query Efficiency:** Optimize queries based on access patterns.

Q112. How do you handle a one-to-many relationship in MongoDB?

Answer:

1. **Embedding:** Use an array to store related documents.
    - Example: A blog post with multiple comments.

```
{
  "postId": "123",
  "comments": [
    { "user": "Mahesh", "text": "Great post!" },
    { "user": "Hema", "text": "Thanks for sharing." }
  ]
}
```

2. **Referencing:** Store ObjectId references in an array.
    - Example:

```
{
  "postId": "123",
  "comments": [ObjectId("comment1"), ObjectId("comment2")]
}
```

3. **Trade-Offs:**
    - Embed for few, frequently accessed related items.
    - Reference for large, infrequently accessed related items.
4. **Indexes:** Use indexes on referenced fields for faster lookups.
5. **Denormalization:** Duplicate data if it improves query performance.

Q113. How do you model a many-to-many relationship in MongoDB?

Answer:

1. **Intermediate Collection:** Use a linking collection to store relationships.
    - Example:

```
{
  "studentId": ObjectId("student123"),
  "courseId": ObjectId("course456")
}
```

2. **Direct Referencing:** Store arrays of references in each collection.
    - Example: Students reference courses, and courses reference students.
3. **Performance Consideration:**
    - Use intermediate collections for scalable relationships.
    - Limit array sizes for better performance.
4. **Indexes:** Add compound indexes on the intermediate collection for faster queries.
5. **Use Case:** Suitable for scenarios like tagging, where relationships can grow dynamically.

Indexing and Performance Tuning:

Q114. What is indexing in MongoDB, and why is it important?

Answer:
Indexing in MongoDB improves the speed and efficiency of query operations.

1. **Faster Query Execution:** Reduces the time taken to locate data in collections.
2. **Default Index:** Each collection has a default index on the _id field.
3. **Custom Indexes:** Create additional indexes on frequently queried fields.
4. **Reduced Disk I/O:** Minimizes the number of documents scanned for queries.
5. **Optimization:** Essential for high-performance applications with large datasets.
6. **Trade-Off:** Increases write operation overhead due to index maintenance.

Q115. What are the different types of indexes in MongoDB?

Answer:

1. **Single Field Index:**
   - Created on a single field for basic query optimization.
   - Example:

```
db.collection.createIndex({ name: 1 });
```

2. **Compound Index:**
   - Combines multiple fields into a single index.
   - Example:

```
db.collection.createIndex({ name: 1, age: -1 });
```

3. **Multikey Index:**
   - Indexes array fields for queries on array elements.
4. **Text Index:**
   - Enables full-text search capabilities.
   - Example:

```
db.collection.createIndex({ description: "text" });
```

5. **TTL (Time-to-Live) Index:**
   - Automatically deletes documents after a specified time.
   - Example:

```
db.collection.createIndex({ createdAt: 1 }, { expireAfterSeconds: 3600 });
```

6. **Sparse and Partial Indexes:**
   - Indexes only documents with the indexed field present.

Q116. How do compound indexes work, and when should you use them?

Answer:

1. **Definition:** Combines multiple fields into a single index to optimize queries involving those fields.
2. **Order Matters:** The order of fields affects query performance.
3. **Example:**

```
db.collection.createIndex({ name: 1, age: -1 });
```

4. **Use Case:** Ideal for queries filtering or sorting by multiple fields.
    - Query: db.collection.find({ name: "Mahesh", age: { $gt: 25 } });
5. **Partial Coverage:** Supports queries on the prefix fields (name in the above example).
6. **Best Practices:** Avoid redundant compound indexes that overlap with existing indexes.

Q117. What is the role of the explain() method in MongoDB?

Answer:

1. **Query Analysis:** Provides insights into how a query is executed.
2. **Execution Plan:** Shows whether a query uses indexes or performs a full collection scan.
3. **Example:**

```
db.collection.find({ name: "Hema" }).explain("executionStats");
```

4. **Performance Metrics:** Includes fields like totalKeysExamined and totalDocsExamined.
5. **Index Debugging:** Identifies queries that could benefit from indexing.
6. **Optimization:** Helps refine queries for better performance.

Q118. What are some common performance tuning techniques in MongoDB?

Answer:

1. **Indexing:** Create indexes for fields frequently used in filters, sorting, or joins.
2. **Sharding:** Distribute large datasets across multiple servers for scalability.
3. **Query Optimization:** Use efficient query patterns to reduce document scans.
4. **Projection:** Retrieve only required fields to reduce data transfer.

```
db.collection.find({}, { name: 1, age: 1 });
```

5. **Aggregation Pipeline:** Process data efficiently using stages like $match and $group.
6. **Monitor Performance:** Use tools like MongoDB Atlas or db.serverStatus() for metrics.

Q119. How do TTL (Time-to-Live) indexes work in MongoDB?

Answer:

1. **Definition:** Automatically deletes documents after a specified time.
2. **Use Case:** Ideal for expiring session tokens, logs, or cache data.
3. **Example:**

```
db.collection.createIndex({ createdAt: 1 }, { expireAfterSeconds: 3600 });
```

4. **Automation:** Removes the need for manual cleanup operations.
5. **Field Requirement:** Works only with fields storing Date objects.
6. **Limitations:** TTL indexes cannot be used for querying data.

Node.js, Express.js and MongoDB 150+ Theory Interview Questions

Q120. How do compound indexes handle sort operations?

Answer:

- **Definition:** Compound indexes optimize queries that filter and sort by multiple fields.

```
db.collection.createIndex({ field1: 1, field2: -1 });
```

- **Key Features:**
  - Supports sorting based on the order of indexed fields.
  - Can only sort on the prefix fields of the compound index.
- **Example Query:**

```
db.collection.find().sort({ field1: 1, field2: -1 });
```

Aggregation Framework:

Q121. What is the Aggregation Framework in MongoDB?

Answer:
The Aggregation Framework in MongoDB is a powerful tool used for data processing and transformation.

1. **Definition:** It processes data using stages in a pipeline for querying and transforming collections.
2. **Pipeline Stages:** Performs operations like filtering, grouping, and sorting in a sequential manner.
3. **Powerful Analytics:** Enables real-time analytics and data transformation.
4. **Common Stages:** $match, $group, $project, $sort, $limit, $unwind.
5. **Performance:** Optimized for efficient data processing by leveraging indexes.
6. **Use Case:** Suitable for generating reports, analytics, and summaries.

Q122. What are the key stages in an aggregation pipeline?

Answer:

1. **$match:** Filters documents to pass only those matching the criteria.
   - Example: { $match: { status: "active" } }.
2. **$group:** Groups documents by a specific field and performs operations like sum or average.
   - Example: { $group: { _id: "$category", total: { $sum: 1 } } }.
3. **$project:** Reshapes documents to include or exclude fields.
   - Example: { $project: { name: 1, age: 1 } }.
4. **$sort:** Orders documents by specified fields.
   - Example: { $sort: { age: -1 } }.
5. **$limit:** Limits the number of documents in the output.
   - Example: { $limit: 5 }.
6. **$unwind:** Deconstructs arrays into separate documents.

- Example: { $unwind: "$tags" }.

Q123. How does the $match stage work, and why is it important?

Answer:

1. **Definition:** Filters documents to pass only those meeting specified criteria.
2. **Query-Like Syntax:** Uses the same syntax as find() queries.

```
{ $match: { status: "active" } }
```

3. **Performance Boost:** Filters data early in the pipeline, reducing processing overhead.
4. **Index Usage:** Leverages indexes for faster filtering.
5. **Use Case:** Ideal for narrowing down large datasets before further processing.

Q124. What is the role of $group in the aggregation framework?

Answer:

1. **Definition:** Groups documents by a specified key and applies aggregate functions.
2. **Key Operations:** Supports functions like $sum, $avg, $max, $min, and $push.
3. **Example:**

```
{ $group: { _id: "$category", totalSales: { $sum: "$amount" } } }
```

4. **Complex Grouping:** Allows nested grouping for hierarchical data processing.
5. **Use Case:** Commonly used for generating summaries, such as totals or averages.

Q125. What is $project, and how is it used in MongoDB?

Answer:

1. **Definition:** Shapes the output documents by including, excluding, or modifying fields.
2. **Include/Exclude Fields:** Specify which fields to include or remove.

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

3. **Computed Fields:** Add new fields based on expressions.

```
{
  $project: {
    fullName: {
      $concat: ["$firstName", " ", "$lastName"];
    }
  }
}
```

4. **Restructure Data:** Modify field names or organize fields hierarchically.
5. **Use Case:** Prepares data for presentation or further processing.

Q126. What are some advanced features of the aggregation framework?

Answer:

1. **Faceted Aggregation:** Perform multiple aggregations in a single pipeline using $facet.

```
{
  $facet: {
    totalSales: [{ $group: { _id: null, total: { $sum: "$amount" } } }],
    productCount: [{ $count: "productCount" }]
  }
}
```

2. **Conditional Logic:** Use $cond or $switch for conditional processing.
3. **Array Processing:** Manipulate arrays with $unwind, $arrayElemAt, and $filter.
4. **Geospatial Aggregation:** Process geospatial data with operators like $geoNear.
5. **Performance Tuning:** Combine $match and $sort early to optimize pipelines.

Q127. What is $unwind, and when should you use it?

Answer:

1. **Definition:** Deconstructs arrays into separate documents for each element.
2. **Example:**

```
{
  $unwind: "$tags";
}
```

3. **Flattening Data:** Converts array fields into individual documents for easier processing.
4. **Preserving Fields:** Use { preserveNullAndEmptyArrays: true } to keep empty arrays.
5. **Use Case:** Analyze or filter individual elements in an array.

Q128. How do you perform a multi-stage aggregation pipeline?

Answer:

1. **Definition:** Combines multiple stages sequentially to process data.
2. **Example Pipeline:**

```
db.collection.aggregate([
  { $match: { status: "active" } },
  { $group: { _id: "$category", totalSales: { $sum: "$amount" } } },
  { $sort: { totalSales: -1 } },
```

```
  { $limit: 5 },
]);
```

3. **Stage Order:** Arrange stages strategically for optimal performance.
4. **Performance Considerations:** Use $match and $project early to reduce processing load.
5. **Output:** Produces aggregated results based on the specified stages.

Q129. What is $lookup in MongoDB, and why is it used?

Answer:
$lookup is an aggregation stage in MongoDB that performs a left outer join to combine documents from two collections.

1. **Definition:** Joins documents from a foreign collection based on a specified field.
2. **Join Type:** Performs a left outer join, retaining all documents from the primary collection.
3. **Aggregation Stage:** Used within an aggregation pipeline.
4. **Data Enrichment:** Combines related data from multiple collections into one output.
5. **Use Case:** Ideal for one-to-many relationships like orders and customers.
6. **Example Query:** Joins orders with customer details.

Q130. How does the $lookup stage work in MongoDB?

Answer:

1. **Syntax:**

```
{
  $lookup: {
    from: "foreignCollection",
    localField: "fieldInPrimary",
    foreignField: "fieldInForeign",
    as: "resultField"
  }
}
```

2. **Parameters:**
   - from: The foreign collection to join with.
   - localField: The field in the primary collection.
   - foreignField: The field in the foreign collection.
   - as: The name of the new array field to store the joined documents.
3. **Output:** Adds a new array field to each document in the primary collection.
4. **Left Outer Join:** Keeps unmatched documents in the primary collection with an empty array.
5. **Use Case:** Combines related data for queries or reports.

Q131. Provide an example of a $lookup query.

Answer:

1. **Collections:**
    - **Orders Collection:**

```
{ "_id": 1, "customerId": 101, "total": 500 }
{ "_id": 2, "customerId": 102, "total": 300 }
```

    - **Customers Collection:**

```
{ "_id": 101, "name": "Mahesh" }
{ "_id": 102, "name": "Hema" }
```

2. **Query:**

```
db.orders.aggregate([
  {
    $lookup: {
      from: "customers",
      localField: "customerId",
      foreignField: "_id",
      as: "customerDetails",
    },
  },
]);
```

3. **Output:**

```
{
  "_id": 1,
  "customerId": 101,
  "total": 500,
  "customerDetails": [{ "_id": 101, "name": "Mahesh" }]
}
```

Q132. What is the difference between $lookup and relational database joins?

Answer:

1. **Join Type:**
    - $lookup: Performs a left outer join.
    - SQL: Supports inner, left, right, and full joins.
2. **Data Storage:**
    - $lookup: Operates on collections within MongoDB.
    - SQL: Operates on tables.
3. **Query Style:**
    - $lookup: Part of an aggregation pipeline.
    - SQL: Directly in the query using JOIN.
4. **Output:**

- $lookup: Produces an array of matching documents.
- SQL: Produces rows combining fields from joined tables.

5. **Flexibility:**
   - $lookup: Works seamlessly with MongoDB's document model.

Q133. What is $lookup with a pipeline, and how does it work?

Answer:

1. **Definition:** Uses an aggregation pipeline for more complex lookups.
2. **Syntax:**

```
{
  $lookup: {
    from: "foreignCollection",
    let: { localFieldVar: "$localField" },
    pipeline: [
      { $match: { $expr: { $eq: ["$foreignField", "$$localFieldVar"] } } }
    ],
    as: "resultField"
  }
}
```

3. **Advanced Queries:** Allows filtering, sorting, and projecting data in the foreign collection.
4. **Dynamic Joins:** Use let to pass variables from the primary collection to the pipeline.
5. **Example Use Case:** Filter related data based on multiple conditions.

Q134. What is the difference between $facet and $bucket in the aggregation framework?

Answer:

- **$facet:** Allows multiple aggregation pipelines to process the same dataset concurrently and outputs a single document with results from each pipeline.

```
{
  $facet: {
    priceBuckets: [{ $bucket: { groupBy: "$price", boundaries: [0, 50, 100], default: "Other" }
}],
    productCount: [{ $count: "total" }]
  }
}
```

- **$bucket:** Divides input documents into specified groups (buckets) based on a field or expression.

```
{
```

```
  $bucket: { groupBy: "$price", boundaries: [0, 50, 100], default: "Other" }
}
```

- **Key Difference:** $facet runs multiple independent pipelines, while $bucket groups data into specified ranges.

Q135. How does $merge stage work in MongoDB, and when should you use it?

Answer:

- **Definition:** $merge writes the results of an aggregation pipeline into an existing collection or creates a new one.

```
{
  $merge: {
    into: "targetCollection",
    on: "_id",
    whenMatched: "merge",
    whenNotMatched: "insert"
  }
}
```

- **Use Cases:**
  - Combining aggregation results with existing data.
  - Incremental updates to collections.

Q136. How does $addFields differ from $project?

Answer:

- **$addFields:** Adds or modifies fields in documents without removing existing fields.

```
{
  $addFields: {
    fullName: {
      $concat: ["$firstName", " ", "$lastName"];
    }
  }
}
```

- **$project:** Controls which fields to include, exclude, or modify in the output.

```
{ $project: { firstName: 1, lastName: 1, fullName: { $concat: ["$firstName", " ",
"$lastName"] } } }
```

- **Key Difference:** $addFields only adds/modifies, whereas $project reshapes the document.

Node.js, Express.js and MongoDB 150+ Theory Interview Questions

Advanced MongoDB Features:

Q137. What are some advanced features of MongoDB that make it a powerful database?

Answer:

1. **Aggregation Framework:** Enables advanced data processing and analysis using pipeline stages.
2. **Sharding:** Distributes large datasets across multiple servers for horizontal scalability.
3. **Replication:** Ensures high availability and data redundancy with replica sets.
4. **Transactions:** Supports multi-document ACID transactions for complex operations.
5. **Full-Text Search:** Provides text indexes for efficient search queries.
6. **Geospatial Queries:** Handles geospatial data for location-based applications.

Q138. What are MongoDB multi-document ACID transactions, and how do they work?

Answer:

1. **Definition:** Guarantees atomicity, consistency, isolation, and durability for multi-document operations.
2. **Use Cases:** Banking transactions, inventory updates, and other critical operations.
3. **Start a Transaction:**

```
const session = client.startSession();
session.startTransaction();
```

4. **Commit/Rollback:** Commit successful transactions or rollback on failure.

```
await session.commitTransaction();
await session.abortTransaction();
```

5. **Replica Set Requirement:** Transactions work only on replica set or sharded cluster deployments.

Q139. What are geospatial queries in MongoDB, and how are they used?

Answer:

1. **Definition:** Handles location-based data with geospatial indexes.
2. **Index Creation:**

```
db.places.createIndex({ location: "2dsphere" });
```

3. **Query Example:**

```
db.places.find({
```

```
  location: {
    $near: {
      $geometry: { type: "Point", coordinates: [40.7128, -74.006] },
      $maxDistance: 5000,
    },
  },
});
```

4. **Supported Queries:** $near, $geoWithin, $geoIntersects.
5. **Use Cases:** Maps, ride-sharing apps, and location-based services.

Q140. What is time series data, and how does MongoDB support it?

Answer:

1. **Definition:** Handles time-stamped data, such as logs or IoT sensor readings.
2. **Time Series Collections:** Optimized for storing and querying time-based data.

```
db.createCollection("metrics", {
  timeseries: { timeField: "timestamp", metaField: "metadata" },
});
```

3. **Efficient Storage:** Reduces storage usage with compression.
4. **Query Optimization:** Provides faster queries for time ranges.
5. **Use Cases:** Monitoring systems, stock market analysis, and IoT applications.

Q141. What are data encryption and security features in MongoDB?

Answer:

1. **Encryption at Rest:** Secures data stored on disk using encryption keys.
2. **Encryption in Transit:** Protects data transmitted over the network with TLS/SSL.
3. **Role-Based Access Control (RBAC):** Grants permissions based on user roles.
4. **Auditing:** Tracks access and changes to data for compliance.
5. **Key Management:** Integrates with external key management systems (KMS).

Backup, Restore, and Deployment:

Q142. What are the primary methods for backing up data in MongoDB?

Answer:

1. **mongodump Tool:** Creates binary backups of databases and collections.
    - Example:

```
mongodump --db myDatabase --out /backup/path
```

2. **File System Snapshot:** Takes snapshots of the database files for faster backups.
3. **Atlas Backups:** MongoDB Atlas provides automated and point-in-time backups.
4. **Replication:** Use replica sets to ensure data redundancy, providing a natural backup layer.
5. **Cloud Storage:** Integrate with AWS, Azure, or GCP for backup storage.
6. **Incremental Backups:** Capture only the changes since the last backup to save time and space.

Q143. How does the mongodump and mongorestore process work in MongoDB?

Answer:

1. **mongodump:** Creates a BSON-formatted backup of the database.
    - Example:

```
mongodump --db myDatabase --out /backup/path
```

2. **mongorestore:** Restores the BSON data to a MongoDB instance.
    - Example:

```
mongorestore --db myDatabase /backup/path/myDatabase
```

3. **Selective Backup:** Backup specific collections using the --collection flag.
4. **Authentication:** Use --username and --password for secure backups.
5. **Compatibility:** Ensure version compatibility between backup and restore tools.

Q144. What are the different deployment architectures in MongoDB?

Answer:

1. **Standalone Deployment:** A single server running a MongoDB instance.
    - Use Case: Development and testing environments.
2. **Replica Set:** A group of MongoDB servers providing redundancy and high availability.
    - Use Case: Production systems needing fault tolerance.
3. **Sharded Cluster:** Distributes data across multiple shards for horizontal scalability.
    - Use Case: Applications handling large datasets or high query volumes.
4. **Cloud Deployment:** MongoDB Atlas provides a fully managed database in the cloud.
    - Use Case: Scalability and ease of management.
5. **Hybrid Deployment:** Combines on-premises and cloud MongoDB instances for flexibility.

Q145. How do you ensure a smooth restore process in MongoDB?

Answer:

1. **Verify Backup:** Test backups regularly to ensure they are complete and functional.
2. **Restore Command:** Use mongorestore with appropriate flags for the environment.

3. **Indexes:** Rebuild or verify indexes after the restore to optimize query performance.
4. **Test in Staging:** Restore backups in a staging environment before production.
5. **Document Restore Steps:** Maintain clear documentation for restore processes.
6. **Authentication:** Use secure credentials during the restore process.

Q146. What is the difference between logical and physical backups in MongoDB?

Answer:

1. **Logical Backups:**
   - Created using tools like mongodump.
   - Data is stored in BSON format.
   - Pros: Portable and compatible across versions.
   - Cons: Slower for large datasets.
2. **Physical Backups:**
   - Captures raw data files and journals.
   - Requires file system-level snapshots.
   - Pros: Faster for large datasets.
   - Cons: Tied to specific MongoDB versions and configurations.
3. **Use Cases:**
   - Logical: For smaller databases or migrations.
   - Physical: For faster backups in large-scale deployments.

Q147.How does MongoDB handle point-in-time recovery?

Answer:

1. **WiredTiger Storage Engine:** Supports snapshots for point-in-time backups.
2. **Journal Files:** Combines data files with journal files to restore to a specific point.
3. **Atlas Backups:** Provides continuous backups with point-in-time recovery options.
4. **Oplog Replay:** Replay operations from the oplog for granular recovery.
5. **Use Case:** Ideal for restoring data after accidental deletions or corruption.

Q148. What are the key considerations for deploying MongoDB in production?

Answer:

1. **Use Replica Sets:** Ensure high availability and data redundancy.
2. **Enable Authentication:** Secure the database with user authentication and roles.
3. **Monitor Performance:** Use tools like MongoDB Compass or Atlas for monitoring.
4. **Index Optimization:** Create indexes for frequently queried fields.
5. **Regular Backups:** Schedule automated backups for disaster recovery.
6. **Scalability:** Plan for sharding if data size is expected to grow significantly.

Q149. What are the advantages of using MongoDB Atlas for backup and deployment?

Answer:

1. **Automated Backups:** Offers snapshot and point-in-time recovery options.
2. **High Availability:** Manages replica sets and sharded clusters automatically.

3. **Scalability:** Provides elastic scaling for data and workloads.
4. **Global Clusters:** Deploys clusters across multiple regions for low latency.
5. **Security:** Includes built-in encryption, auditing, and role-based access control.
6. **Ease of Use:** Simplifies management with an intuitive UI and APIs.

Q150. What is oplog, and how is it used in backup and restore processes?

Answer:

1. **Definition:** The oplog (operations log) records all write operations in a replica set.
2. **Change Tracking:** Used for replicating changes to secondary members.
3. **Point-in-Time Restore:** Replay oplog entries to restore data to a specific time.
4. **Continuous Backups:** Enables incremental backups by capturing oplog entries.
5. **Use Case:** Sync data between nodes or recover from partial backups.

Q151. What are best practices for managing MongoDB backups and deployments?

Answer:

1. **Automate Backups:** Use tools or scripts to schedule regular backups.
2. **Test Restore Process:** Regularly validate that backups can be restored successfully.
3. **Monitor Resources:** Ensure sufficient storage and compute power for backups.
4. **Document Procedures:** Maintain clear guidelines for backup and restore processes.
5. **Use Cloud Services:** Leverage MongoDB Atlas for managed backups and deployments.
6. **Secure Backups:** Encrypt backup files and restrict access to authorized users.

Node.js, Express.js and MongoDB 10+ Interview Task and Solutions

1. Task: Build a RESTful API in Express.js for a "Todo" application with endpoints to create, fetch, update, and delete todos.

Solution:

```
const express = require("express");
const app = express();

app.use(express.json());

let todos = [];

app.post("/todos", (req, res) => {
  const { id, task } = req.body;
  todos.push({ id, task });
  res.status(201).send("Todo added.");
});

app.get("/todos", (req, res) => {
  res.json(todos);
});

app.put("/todos/:id", (req, res) => {
  const { id } = req.params;
  const { task } = req.body;
  const todo = todos.find((t) => t.id === id);
  if (todo) {
    todo.task = task;
    res.send("Todo updated.");
  } else {
    res.status(404).send("Todo not found.");
  }
});

app.delete("/todos/:id", (req, res) => {
  const { id } = req.params;
  todos = todos.filter((t) => t.id !== id);
  res.send("Todo deleted.");
});

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

Steps:

1. **Set Up Express.js:**
   - Install Express.js and import it.
   - Initialize an Express application.

```
const express = require("express");
```

```
const app = express();
```

2. **Middleware for JSON Parsing:**
    - Use express.json() middleware to parse incoming JSON payloads.

```
app.use(express.json());
```

3. **Create a Global todos Array:**
    - Use an in-memory array to store todos.

```
let todos = [];
```

4. **Define Endpoints:**
    - **POST /todos:** Add a new todo.
        - Extract id and task from the request body.
        - Push the new todo to the todos array.

```
app.post("/todos", (req, res) => {
  const { id, task } = req.body;
  todos.push({ id, task });
  res.status(201).send("Todo added.");
});
```

    - **GET /todos:** Fetch all todos.

```
app.get("/todos", (req, res) => {
  res.json(todos);
});
```

    - **PUT /todos/:id:** Update a todo by id.
        - Find the todo with the matching id and update the task.

```
app.put("/todos/:id", (req, res) => {
  const { id } = req.params;
  const { task } = req.body;
  const todo = todos.find((t) => t.id === id);
  if (todo) {
    todo.task = task;
    res.send("Todo updated.");
  } else {
    res.status(404).send("Todo not found.");
  }
});
```

    - **DELETE /todos/:id:** Remove a todo by id.
        - Filter out the todo with the matching id from the array.

```
app.delete("/todos/:id", (req, res) => {
  const { id } = req.params;
  todos = todos.filter((t) => t.id !== id);
  res.send("Todo deleted.");
});
```

5. **Start the Server:**
    - Listen on port 3000.

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

2. Task: Create a middleware in Express.js to log the request method and URL for every incoming request.

Solution:

```
const express = require("express");
const app = express();

// Middleware
app.use((req, res, next) => {
  console.log(`Request Method: ${req.method}, URL: ${req.url}`);
  next();
});

app.get("/", (req, res) => {
  res.send("Hello, Middleware!");
});

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

1. Import Express Module:

```
const express = require("express");
```

- The express module is imported to create an Express application.
- Express.js is a web application framework for Node.js that simplifies server and routing logic.

2. Create an Express Application

```
const app = express();
```

- The app object is created by calling express().
- This object is used to define routes, middleware, and application behavior.

Node.js, Express.js and MongoDB 10+ Interview Task and Solutions

3. Middleware Definition

```
app.use((req, res, next) => {
  console.log(`Request Method: ${req.method}, URL: ${req.url}`);
  next();
});
```

- **Middleware:** A function that sits between the client request and the server response. It can modify the request or response objects or terminate the request-response cycle.
- **app.use()**: Adds middleware to the application. This middleware logs the HTTP request method (e.g., GET, POST) and the requested URL.
- **req.method**: Provides the HTTP method of the request (e.g., GET, POST).
- **req.url**: Contains the URL path of the request.
- **next()**: Calls the next middleware function in the stack. Without next(), the request will hang and not proceed further.

4. Define a Route

```
app.get("/", (req, res) => {
  res.send("Hello, Middleware!");
});
```

- **app.get()**: Defines a route that handles GET requests to the root URL (/).
- **Request Handler:** The callback function (req, res) handles the request and sends a response.
    - req: The request object, containing data about the HTTP request.
    - res: The response object, used to send a response back to the client.
- **Response:** The server sends the message "Hello, Middleware!" to the client when this route is accessed.

5. Start the Server

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

- **app.listen(3000)**: Starts the server on port 3000 and listens for incoming requests.
- **Callback Function:** Logs a message to the console when the server starts successfully.

Flow of Execution

1. When a request is made, the middleware runs first, logging the request method and URL to the console.
    - Example Log:

```
Request Method: GET, URL: /
```

2. If the URL matches the / route, the request handler sends the response "Hello, Middleware!" to the client.
3. If the URL does not match any route, the request will result in a 404 Not Found error since no other routes or error-handling middleware are defined.

Node.js, Express.js and MongoDB 10+ Interview Task and Solutions

Output Example

- When you visit http://localhost:3000/ in a browser or use a tool like Postman:
    - **Console Output:**

```
Request Method: GET, URL:
```

    - **Client Response:**

```
Hello, Middleware!
```

Task 3: Write a MongoDB query to find all users who registered in the last 30 days.

Solution:

```
const { MongoClient } = require("mongodb");
const uri = "mongodb://localhost:27017";
const client = new MongoClient(uri);

async function getRecentUsers() {
  await client.connect();
  const db = client.db("testdb");
  const users = db.collection("users");

  const thirtyDaysAgo = new Date();
  thirtyDaysAgo.setDate(thirtyDaysAgo.getDate() - 30);

  const recentUsers = await users
    .find({ createdAt: { $gte: thirtyDaysAgo } })
    .toArray();
  console.log(recentUsers);

  await client.close();
}

getRecentUsers();
```

Steps:

1. **Connect to MongoDB:**
    - Use MongoClient to connect to the MongoDB instance.

```
const { MongoClient } = require("mongodb");
const uri = "mongodb://localhost:27017";
const client = new MongoClient(uri);
```

2. **Determine the Date Range:**
    - Calculate the date 30 days ago.

```
const thirtyDaysAgo = new Date();
thirtyDaysAgo.setDate(thirtyDaysAgo.getDate() - 30);
```

3. **Query the Collection:**
    - Use the $gte operator to find documents with createdAt greater than or equal to thirtyDaysAgo.

```
const recentUsers = await users
  .find({ createdAt: { $gte: thirtyDaysAgo } })
  .toArray();
```

4. **Print the Results:**
    - Log the result to the console and close the connection.

```
console.log(recentUsers);
await client.close();
```

Task 4: Use the aggregation framework to calculate the total sales for each product category.

Solution:

```
const { MongoClient } = require("mongodb");
const uri = "mongodb://localhost:27017";
const client = new MongoClient(uri);

async function calculateTotalSales() {
  await client.connect();
  const db = client.db("ecommerce");
  const sales = db.collection("sales");

  const result = await sales
    .aggregate([
      { $group: { _id: "$category", totalSales: { $sum: "$amount" } } },
      { $sort: { totalSales: -1 } },
    ])
    .toArray();

  console.log(result);
  await client.close();
}

calculateTotalSales();
```

Steps:

1. **Connect to MongoDB:**
    - Use MongoClient to connect to the database.

```
const { MongoClient } = require("mongodb");
const uri = "mongodb://localhost:27017";
const client = new MongoClient(uri);
```

2. **Aggregation Pipeline:**
    - Define the pipeline stages:
        - **$group:** Group by category and sum the amount field.
        - **$sort:** Sort the results by totalSales in descending order.

```
const pipeline = [
  { $group: { _id: "$category", totalSales: { $sum: "$amount" } } },
  { $sort: { totalSales: -1 } },
];
```

3. **Execute Aggregation:**
    - Run the aggregate() method with the pipeline.

```
const result = await sales.aggregate(pipeline).toArray();
console.log(result);
```

4. **Close the Connection:**
    - Close the client after execution.

```
await client.close();
```

Task 5: Create an API to fetch paginated user data from MongoDB, including the current page, total records, and total pages.

Solution:

```
app.get("/users", async (req, res) => {
  const page = parseInt(req.query.page) || 1;
  const limit = parseInt(req.query.limit) || 10;
  const skip = (page - 1) * limit;

  const users = await usersCollection.find().skip(skip).limit(limit).toArray();
  const totalUsers = await usersCollection.countDocuments();

  res.json({
    totalRecords: totalUsers,
    page,
    totalPages: Math.ceil(totalUsers / limit),
    users,
  });
});
```

Explanation:

1. **Pagination Parameters:**
   - page: The current page number (default is 1).
   - limit: The number of records per page (default is 10).
   - skip: Calculated as (page - 1) * limit to skip the required number of records.
2. **Query Execution:**
   - The find method fetches records, applying skip and limit for pagination.
3. **Total Records:**
   - countDocuments provides the total number of records in the collection.
4. **Response:**
   - Includes the total records, current page, total pages, and the paginated users.

Task 6: Create an API to add products with a name and description, and another API to search for products using a text query.

Solution:

```
// Create product
app.post("/products", async (req, res) => {
  const product = req.body;
  const result = await productsCollection.insertOne(product);
  res.status(201).json(result);
});

// Create text index
await productsCollection.createIndex({ name: "text", description: "text" });

// Search product
app.get("/search", async (req, res) => {
  const query = req.query.q;
  const results = await productsCollection
    .find({ $text: { $search: query } })
    .toArray();
  res.json(results);
});
```

Explanation:

1. **Create Product:**
   - Adds products with fields like name and description to the MongoDB collection.
2. **Text Index:**
   - A text index is created on the name and description fields to support text-based searches.
3. **Search Query:**
   - The $text operator performs a full-text search on indexed fields.
4. **Response:**
   - Returns all products matching the search query.

Task 7: Create an API to provide real-time notifications for database changes using change streams.

Node.js, Express.js and MongoDB 10+ Interview Task and Solutions

Solution:

```
app.get("/notifications", (req, res) => {
  res.setHeader("Content-Type", "text/event-stream");
  res.setHeader("Cache-Control", "no-cache");
  res.setHeader("Connection", "keep-alive");

  const changeStream = productsCollection.watch();

  changeStream.on("change", (change) => {
    res.write(`data: ${JSON.stringify(change)}\n\n`);
  });

  req.on("close", () => {
    changeStream.close();
    res.end();
  });
});
```

Explanation:

1. **Server-Sent Events (SSE):**
   - Configures HTTP headers for real-time streaming to the client.
2. **Change Stream:**
   - Watches the productsCollection for any changes (insert, update, delete).
3. **Real-Time Data:**
   - Sends the change details to the client as they occur.
4. **Handle Disconnection:**
   - Closes the change stream and ends the response when the client disconnects.

Task 8: Create an API to log in a user and generate a JSON Web Token (JWT) for authentication. Validate the token for accessing protected routes.

Solution:

```
const jwt = require("jsonwebtoken");
const SECRET_KEY = "your_secret_key";

// Login route
app.post("/login", async (req, res) => {
  const { username, password } = req.body;
  const user = await usersCollection.findOne({ username });

  if (user && (await bcrypt.compare(password, user.password))) {
    const token = jwt.sign({ username, role: user.role }, SECRET_KEY, {
      expiresIn: "1h",
    });
    res.json({ token });
  } else {
    res.status(401).send("Invalid username or password");
```

```
  }
});

// Middleware to validate token
function authenticateToken(req, res, next) {
  const token = req.headers["authorization"];
  if (!token) return res.status(403).send("Token is required");

  jwt.verify(token, SECRET_KEY, (err, user) => {
    if (err) return res.status(403).send("Invalid token");
    req.user = user;
    next();
  });
}

// Protected route
app.get("/dashboard", authenticateToken, (req, res) => {
  res.send(`Welcome, ${req.user.username}`);
});
```

Explanation:

1. **Login Endpoint:**
   - Authenticates the user and generates a JWT with username and role as payload.
   - Sets the token expiration to 1 hour.
2. **Token Validation Middleware:**
   - Checks for the presence of a token in the Authorization header.
   - Verifies the token using jwt.verify.
3. **Protected Route:**
   - The /dashboard route is accessible only with a valid token.

Task 9: Create an API to upload files to MongoDB using GridFS and retrieve them by filename.

Solution:

```
const multer = require("multer");
const { GridFsStorage } = require("multer-gridfs-storage");
const storage = new GridFsStorage({ url: "mongodb://localhost:27017/filesdb" });
const upload = multer({ storage });

// Upload file
app.post("/upload", upload.single("file"), (req, res) => {
  res.status(201).send("File uploaded successfully");
});

// Retrieve file
app.get("/files/:filename", async (req, res) => {
```

```
  const { filename } = req.params;
  const bucket = new mongodb.GridFSBucket(client.db("filesdb"));
  bucket.openDownloadStreamByName(filename).pipe(res);
});
```

Explanation:

1. **GridFS:**
   - A MongoDB specification for storing and retrieving large files.
   - Splits files into smaller chunks for storage.
2. **File Upload:**
   - multer-gridfs-storage handles the storage of files in GridFS.
3. **File Retrieval:**
   - Streams the file from GridFS to the client using GridFSBucket.

Task 10 : Add a rate limiter to restrict users to a maximum of 5 requests per minute.

Solution:

```
const rateLimit = require("express-rate-limit");

// Rate limiter middleware
const limiter = rateLimit({
  windowMs: 1 * 60 * 1000, // 1 minute
  max: 5, // Limit each IP to 5 requests per minute
  message: "Too many requests, please try again later.",
});

// Apply the rate limiter to all routes
app.use(limiter);

app.get("/", (req, res) => {
  res.send("Welcome to the API!");
});
```

Explanation:

1. **Rate Limiter Setup:**
   - Limits the number of requests per IP within a 1-minute window.
2. **Global Application:**
   - Applied to all routes using app.use(limiter).
3. **Response to Excessive Requests:**
   - Responds with a 429 Too Many Requests status and a custom message.

Task 11: Create an API to perform bulk insertion of user records, with validation for required fields.

Solution:

```
app.post("/users/bulk", async (req, res) => {
```

```
  const users = req.body;

  // Validate each user
  const invalidUsers = users.filter(
    (user) => !user.username || !user.email || !user.password
  );

  if (invalidUsers.length > 0) {
    return res
      .status(400)
      .json({ error: "Invalid user records", invalidUsers });
  }

  const result = await usersCollection.insertMany(users);
  res.status(201).json(result);
});
```

Explanation:

1. **Bulk Insertion:**
   - Accepts an array of user objects for insertion.
2. **Validation:**
   - Filters out records missing username, email, or password.
3. **Response:**
   - Inserts valid records and returns a success response or an error with invalid records.