Level 1: JavaScript, Node.js, MongoDB, Express.js, JWT

📌 Conceptual Question (1)

1. **The Heart of Node.js:** Explain the Node.js **event loop** in detail. Your explanation should not only define what it is but also differentiate between the **macro-task queue** (e.g., for setTimeout, setInterval) and the **micro-task queue** (e.g., for Promise.then(), process.nextTick()). Describe a practical scenario where a misunderstanding of the execution order between these two queues could lead to a subtle but critical bug in a backend application (e.g., in a financial transaction or a real-time notification system).

💻 Practical Questions (10)

Imagine you are building the backend API for a multi-user blogging platform called "Inkwell."

1. Advanced Middleware - Rate Limiter & Logger:
   Create two pieces of Express middleware from scratch (without using external libraries like express-rate-limit or morgan).
   - **Request Logger**: This middleware should log the incoming request's method, URL, timestamp, and IP address to a file named requests.log. It must handle file writing asynchronously without blocking the event loop.
   - **IP-Based Rate Limiter**: This middleware should limit each IP address to **100 requests per 1-minute window**. If an IP exceeds the limit, the server should respond with a 429 Too Many Requests status code. You'll need to store request timestamps in memory for this.
2. Data Modeling - Schemas & Relationships:
   Using Mongoose, design the schemas for User, Post, and Comment.
   - A User must have a name, email (unique), password (to be hashed), and a role (enum of 'user' or 'admin').
   - A Post must have a title, content, author (a reference to the User model), and tags (an array of strings).
   - A Comment must have text, author (a reference to the User model), and post (a reference to the Post model).
     Justify your decision to use referencing instead of embedding for these relationships, considering data consistency and query patterns.
3. Authentication - JWT with Refresh Tokens:

Implement a complete JWT-based authentication flow.
    - Create a /api/auth/register endpoint that hashes the user's password using a library like bcrypt before saving it to MongoDB.
    - Create a /api/auth/login endpoint that validates credentials and, upon success, returns both an **access token** (JWT, expires in 15 minutes, contains user ID and role) and a **refresh token** (a secure, random string stored in the database, linked to the user). The refresh token should be sent to the client in a secure, httpOnly cookie.
4. Authorization - Role-Based Access Control Middleware:
   Create two separate authorization middlewares.
    - protect: This middleware verifies the access token from the Authorization header. If valid, it decodes the payload, fetches the corresponding user from the database, and attaches it to the req object (as req.user).
    - authorize(...roles): This is a higher-order middleware that accepts an array of roles (e.g., authorize('admin')). It should check if req.user.role is included in the provided roles. If not, it should deny access. Use these middlewares to protect an endpoint so only an 'admin' can access it.
5. Advanced Queries - Aggregation Pipeline:
   Create an API endpoint GET /api/analytics/posts/popular that requires admin authorization. This endpoint should use a MongoDB Aggregation Pipeline to return a list of the top 5 authors who have the highest average number of comments per post. The result should be sorted in descending order and include the author's name and their calculated average.
6. Error Handling - Centralized & Custom:
   Implement a robust, centralized error handling system.
    - Create a custom ErrorHandler class that extends JavaScript's built-in Error class and accepts a message and a statusCode.
    - Create a final middleware in your Express app that catches all errors (including your custom ones). It should format a consistent JSON error response { success: false, error: message } and set the HTTP status code based on the error object.
    - Refactor one of your route handlers to use this system by throwing a new ErrorHandler instance (e.g., throw new ErrorHandler('Post not found', 404)).
7. File Uploads with Streams:
   Create a PUT /api/users/avatar endpoint that allows an authenticated user to upload a profile picture. Instead of using a library like multer to handle the file, use Node.js's native streams. Pipe the incoming request stream directly to a file stream (fs.createWriteStream) to save the image to the server's filesystem. Implement basic validation to ensure the file is an image (check the Content-Type header) and reject files larger than 2MB by listening to the stream's data events.
8. Data Integrity - MongoDB Transactions:
   Imagine a feature where deleting a User account must also delete all Posts and Comments created by that user. Implement the logic for this in a DELETE /api/users/profile endpoint. The entire operation must be atomic. Use a MongoDB session and transaction to ensure that if any of the deletion steps fail, all previous steps are

rolled back, leaving the database in its original state.

9. Async JavaScript - Custom asyncHandler Utility:
   In Express, every async route handler needs a try...catch block to pass errors to the error handling middleware. This is repetitive. Write a higher-order function named asyncHandler that takes a controller function as an argument. This utility function should return a new function that executes the original controller, catches any promise rejections, and passes the error to next(). Show how you would use this utility to simplify your route definitions (e.g., router.get('/', asyncHandler(getAllPosts))).
10. Security Hardening:
    Implement three critical security measures in your Express application.
    - **Input Sanitization**: Use a library like express-mongo-sanitize to prevent NoSQL query injection by stripping characters like $ and . from the request body, query params, etc.
    - **Security Headers**: Use a library like helmet to set various security-related HTTP headers to protect against common attacks like Cross-Site Scripting (XSS) and clickjacking.
    - **CORS**: Configure the cors middleware to only allow requests from a specific frontend URL (e.g., http://localhost:3000) instead of allowing all origins (*).

🔧 Code Review & Debugging (1)

1. **Find the Bugs:** The following code snippet is for a new user registration endpoint. It is riddled with security flaws, race conditions, and poor practices. Review the code, identify at least **five distinct issues**, and then **refactor it** into a secure, robust, and efficient version. Add comments to your refactored code explaining what you fixed and why.

```
JavaScript
// Original Buggy Code
const User = require('../models/User');

app.post('/register', (req, res) => {
  const { email, password, name } = req.body;

  // Check if user exists
  const user = User.findOne({ email: email });

  if (user) {
    res.send('Error: User already exists.');
  } else {
    const newUser = new User({
```

```
    name: name,
    email: email,
    password: password // Storing password in plain text
  });

  newUser.save(); // Asynchronous operation without await or .then()

  res.json({ status: 'User created' }); // Responds before save is complete
 }
});
```