Junior Backend Developer Interview Questions

Tech Stack: Node.js, Express.js, MySQL/PostgreSQL, TypeScript

Table of Contents

Node.js Fundamentals

1. What is Node.js and why is it used?

Answer:

Node.js is a JavaScript runtime built on Chrome's V8 engine that allows JavaScript to run on the server-side. It's used for:

- Building scalable network applications
- Non-blocking I/O operations
- Single programming language (JavaScript) for both frontend and backend
- Large ecosystem (npm)
- Great for I/O-heavy applications

2. What is the Event Loop in Node.js?

Answer:

The Event Loop is the mechanism that handles asynchronous operations in Node.js. It continuously checks the call stack and callback queue, executing callbacks when the stack is empty.

How it works:

Node.js is single-threaded but can handle concurrent operations through the event loop. When asynchronous operations (like file I/O, network requests) are initiated, they're delegated to the system kernel or thread pool, freeing the main thread to continue executing other code.

Event Loop Phases (in order):

1. **Timers**: Executes callbacks scheduled by `setTimeout()` and `setInterval()`

2. **Pending callbacks**: Executes I/O callbacks deferred from the previous cycle
3. **Idle, prepare**: Internal operations (used internally by Node.js)
4. **Poll**: Retrieve new I/O events; execute I/O related callbacks
5. **Check**: Execute `setImmediate()` callbacks
6. **Close callbacks**: Execute close event callbacks (e.g., `socket.on('close')`)

Microtasks vs Macrotasks:

- **Microtasks** (Promises, process.nextTick): Execute immediately after current operation, before moving to next phase
- **Macrotasks** (setTimeout, setImmediate): Execute in their respective event loop phases

```
console.log('Start'); // 1. Synchronous - executes
immediately

setTimeout(() => {
  console.log('Timeout'); // 4. Macrotask - goes to
timers phase
}, 0);

Promise.resolve().then(() => {
  console.log('Promise'); // 3. Microtask - executes
after sync code, before macrotasks
});

console.log('End'); // 2. Synchronous - executes
```

```
immediately

// Output: Start, End, Promise, Timeout
// Why? Synchronous code runs first, then microtasks
(Promises), then macrotasks (setTimeout)
```

Key Points:

- The event loop allows Node.js to perform non-blocking I/O operations despite JavaScript being single-threaded
- Understanding the event loop is crucial for optimizing async code and avoiding blocking operations
- Always use async operations for I/O to prevent blocking the event loop

3. What is the difference between `require()` and `import`?

Answer:

- **require()**: CommonJS module system, synchronous, dynamic loading
- **import**: ES6 module system, can be tree-shaken, static analysis

```
// CommonJS
const express = require('express');

// ES6
import express from 'express';
```

4. What is `package.json` and what is its purpose?

Answer:

`package.json` is a manifest file that contains:

- Project metadata (name, version, description)
- Dependencies and devDependencies
- Scripts (start, test, build)
- Configuration for various tools

5. What are Streams in Node.js?

Answer:

Streams are objects for handling reading/writing data continuously in chunks rather than loading everything into memory at once. This is especially useful for large files or real-time data.

Why use Streams?

- **Memory Efficient**: Process data chunk by chunk instead of loading entire file into memory
- **Time Efficient**: Start processing data as soon as first chunk arrives (don't wait for entire file)
- **Composable**: Chain multiple stream operations using `.pipe()`

Types of Streams:

1. **Readable**: Read data from a source (e.g., `fs.createReadStream`, HTTP request)
2. **Writable**: Write data to a destination (e.g., `fs.createWriteStream`, HTTP response)

3. **Duplex**: Both read and write (e.g., `net.Socket`, TCP sockets)
4. **Transform**: Modify data while reading/writing (e.g., `zlib`, compression)

Basic Example:

```
import fs from 'fs';

// Instead of reading entire file into memory (BAD for
large files)
// const data = fs.readFileSync('large-file.txt'); //
Loads entire file!

// Use streams to process data in chunks (GOOD for
large files)
const readStream = fs.createReadStream('input.txt', {
  encoding: 'utf8',
  highWaterMark: 16 * 1024 // 16KB chunks (default:
64KB)
});

const writeStream = fs.createWriteStream('output.txt');

// Pipe automatically handles backpressure and errors
readStream.pipe(writeStream);

// Event-based approach for more control
readStream.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes`);
  // Process chunk here
});
```

```
readStream.on('end', () => {
  console.log('Finished reading file');
});

readStream.on('error', (err) => {
  console.error('Error reading file:', err);
});
```

Practical Example - Transform Stream:

```
import { Transform } from 'stream';

// Create a transform stream to convert text to
uppercase
const upperCaseTransform = new Transform({
  transform(chunk, encoding, callback) {
    // Process each chunk
    this.push(chunk.toString().toUpperCase());
    callback();
  }
});

// Chain multiple operations
fs.createReadStream('input.txt')
  .pipe(upperCaseTransform)
  .pipe(fs.createWriteStream('output.txt'));

// Example: This can process a 1GB file using only
~64KB of memory at a time!
```

6. What is `process` in Node.js?

Answer:

`process` is a global object providing information about the current Node.js process:

- `process.env`: environment variables
- `process.argv`: command-line arguments
- `process.exit()`: exit the process
- `process.cwd()`: current working directory

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

Answer:

- **process.nextTick()**: Executes before the next event loop iteration (microtask queue)
- **setImmediate()**: Executes on the next event loop iteration (check phase)

```
setImmediate(() => console.log('Immediate'));
process.nextTick(() => console.log('Next Tick'));
// Output: Next Tick, Immediate
```

8. What is middleware in Node.js?

Answer:

Middleware functions have access to request, response, and next function. They can:

- Execute code

- Modify request/response objects
- End request-response cycle
- Call next middleware

9. What is `Buffer` in Node.js?

Answer:

Buffer is a class for handling binary data directly. Used when working with:

- File operations
- Network streams
- Cryptography

```
const buf = Buffer.from('Hello');
console.log(buf.toString()); // 'Hello'
console.log(buf.toString('hex')); // '48656c6c6f'
```

10. What is `cluster` module?

Answer:

The cluster module allows creating child processes (workers) that share the same server port, enabling your Node.js application to utilize all CPU cores for better performance.

Why use Cluster?

Node.js runs in a single thread by default. On a server with 8 CPU cores, a basic Node.js app only uses 1 core (12.5% of available resources). The cluster module lets you spawn multiple processes to utilize all cores.

How it works:

- **Primary process**: Manages worker processes and distributes incoming connections
- **Worker processes**: Handle actual application logic; each runs in its own V8 instance
- Load balancing: The primary process distributes incoming connections across workers using round-robin (by default)

```
import cluster from 'cluster';
import os from 'os';
import express from 'express';

if (cluster.isPrimary) {
  // This code runs once (primary process)
  const numCPUs = os.cpus().length; // e.g., 8 cores
  console.log(`Primary process ${process.pid} is
running`);
  console.log(`Forking ${numCPUs} workers...`);

  // Create a worker for each CPU core
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  // Handle worker crashes
  cluster.on('exit', (worker, code, signal) => {
    console.log(`Worker ${worker.process.pid} died.
Starting new worker...`);
    cluster.fork(); // Restart crashed worker
```

```
  });

} else {
  // This code runs in each worker process
  // Each worker runs the application independently
  const app = express();

  app.get('/', (req, res) => {
    res.send(`Hello from worker ${process.pid}`);
  });

  app.listen(3000, () => {
    console.log(`Worker ${process.pid} started`);
  });
}

// Result: If you have 8 cores, you now have 8 Node.js
processes
// handling requests simultaneously!
```

Benefits:

- **Better CPU utilization**: Use all available cores
- **Higher throughput**: Handle more concurrent requests
- **Fault tolerance**: If one worker crashes, others continue running

Considerations:

- Each worker has its own memory space (no shared variables between workers)
- For shared state, use external storage (Redis, database)

- PM2 or similar process managers often handle clustering automatically

---

Express.js Essentials

11. What is Express.js?

Answer:

Express.js is a minimal and flexible Node.js web application framework providing:

- Robust routing
- HTTP utility methods and middleware
- Quick API development
- Template engine support

12. How do you create a basic Express server?

Answer:

```
import express, { Request, Response } from 'express';

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

app.use(express.json());

app.get('/', (req: Request, res: Response) => {
  res.json({ message: 'Hello World' });
});
```

```
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

13. What are Express middleware and their types?

Answer:

Middleware functions are functions that have access to the request object (`req`), response object (`res`), and the `next` function. They execute in the order they're defined and can:

- Execute any code
- Modify request/response objects
- End the request-response cycle
- Call the next middleware in the stack

Think of middleware as a pipeline: Request → Middleware 1 → Middleware 2 → ... → Route Handler → Response

Types of Middleware:

1. **Application-level Middleware**: Bound to `app` instance, runs for all routes or specific routes

```
// Runs for all routes
app.use((req, res, next) => {
  console.log('Time:', Date.now());
  next(); // MUST call next() to pass control to next middleware
```

```
});

// Runs only for paths starting with /api
app.use('/api', (req, res, next) => {
  console.log('API request');
  next();
});
```

2. **Router-level Middleware**: Bound to `Router` instance, used for modular route handling

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

// Only applies to routes defined in this router
router.use(authMiddleware);
router.get('/profile', (req, res) => { /* ... */ });

app.use('/api/users', router);
```

3. **Error-handling Middleware**: Has 4 parameters (err, req, res, next) - MUST be defined last

```
// Error handler (notice 4 parameters)
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(err.statusCode || 500).json({
    error: err.message
  });
});
```

4. **Built-in Middleware**: Provided by Express

```
app.use(express.json());        // Parse JSON request
bodies
app.use(express.urlencoded({ extended: true })); //
Parse URL-encoded bodies
app.use(express.static('public')); // Serve static
files
```

5. **Third-party Middleware**: External packages

```
import cors from 'cors';
import helmet from 'helmet';
import morgan from 'morgan';

app.use(cors());        // Enable CORS
app.use(helmet());      // Security headers
app.use(morgan('dev')); // HTTP request logger
```

Middleware Execution Flow:

```
app.use((req, res, next) => {
  console.log('1. First middleware');
  next();
});

app.use((req, res, next) => {
  console.log('2. Second middleware');
  next();
});
```

```
app.get('/test', (req, res) => {
  console.log('3. Route handler');
  res.send('Done');
});

// Request to /test outputs:
// 1. First middleware
// 2. Second middleware
// 3. Route handler
```

Important: If you don't call `next()` or send a response, the request will hang!

14. How do you handle different HTTP methods in Express?

Answer:

```
app.get('/users', (req, res) => {});       // Read
app.post('/users', (req, res) => {});      // Create
app.put('/users/:id', (req, res) => {});   // Update
(full)
app.patch('/users/:id', (req, res) => {}); // Update
(partial)
app.delete('/users/:id', (req, res) => {}); // Delete
```

15. What is `req` and `res` in Express?

Answer:

- **req (Request)**: Contains HTTP request information
  - `req.body`: POST data

  - `req.params`: URL parameters
  - `req.query`: Query string
  - `req.headers`: HTTP headers

- **res (Response)**: Used to send HTTP response
  - `res.json()`: Send JSON response
  - `res.status()`: Set status code
  - `res.send()`: Send response
  - `res.render()`: Render view

16. How do you handle route parameters?

Answer:

```
// URL parameters
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  res.json({ userId });
});

// Query parameters
app.get('/search', (req, res) => {
  const { q, page } = req.query;
  res.json({ query: q, page });
});
// URL: /search?q=nodejs&page=1
```

17. What is Router in Express?

Answer:

Router is a mini-application for organizing routes.

```
// routes/users.ts
import { Router } from 'express';
const router = Router();

router.get('/', (req, res) => {});
router.post('/', (req, res) => {});
router.get('/:id', (req, res) => {});

export default router;

// app.ts
import userRoutes from './routes/users';
app.use('/api/users', userRoutes);
```

18. How do you serve static files in Express?

Answer:

```
app.use(express.static('public'));
// Files in 'public' folder are accessible at root URL

app.use('/static', express.static('public'));
// Files accessible at /static/filename
```

19. What is CORS and how to enable it?

Answer:

CORS (Cross-Origin Resource Sharing) allows requests from different origins.

```
import cors from 'cors';

// Enable all CORS requests
app.use(cors());

// Configure CORS
app.use(cors({
  origin: 'https://example.com',
  methods: ['GET', 'POST'],
  credentials: true
}));
```

20. How do you handle 404 errors?

Answer:

```
// Place after all routes
app.use((req, res) => {
  res.status(404).json({ error: 'Route not found' });
});
```

MySQL/PostgreSQL

21. What is the difference between SQL and NoSQL databases?

Answer:

SQL (MySQL, PostgreSQL):

- Structured data with fixed schema
- ACID compliant
- Relations between tables
- Vertical scaling
- Good for complex queries

NoSQL:

- Flexible schema
- Horizontal scaling
- Better for unstructured data

22. What are the main differences between MySQL and PostgreSQL?

Answer:

Feature	MySQL	PostgreSQL
ACID Compliance	InnoDB only	Full
Complex Queries	Good	Excellent
JSON Support	Basic	Advanced
Performance	Fast reads	Balanced

Feature	MySQL	PostgreSQL
Standards Compliance	Moderate	High

23. What are Primary Keys and Foreign Keys?

Answer:

```
-- Primary Key: Unique identifier for each row
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email VARCHAR(255) UNIQUE NOT NULL
);

-- Foreign Key: Links to another table
CREATE TABLE posts (
  id SERIAL PRIMARY KEY,
  user_id INTEGER REFERENCES users(id),
  title VARCHAR(255) NOT NULL
);
```

24. What are SQL JOINs?

Answer:
JOINs are used to combine rows from two or more tables based on a related column between them. Understanding JOINs is crucial for working with relational databases.

Sample Data:

```
-- Users table
| id | name    |
|----|---------|
| 1  | Alice   |
| 2  | Bob     |
| 3  | Charlie |

-- Orders table
| id | user_id | total |
|----|---------|-------|
| 1  | 1       | 100   |
| 2  | 1       | 200   |
| 3  | 2       | 150   |
```

1. **INNER JOIN** - Returns only matching rows from both tables

```
SELECT users.name, orders.total
FROM users
INNER JOIN orders ON users.id = orders.user_id;

-- Result: Only users who have orders
| name  | total |
|-------|-------|
| Alice | 100   |
| Alice | 200   |
| Bob   | 150   |
-- Charlie is excluded (no orders)

-- Use when: You only want records that exist in both
tables
```

2. LEFT JOIN (LEFT OUTER JOIN) - All rows from left table, matching rows from right

```
SELECT users.name, orders.total
FROM users
LEFT JOIN orders ON users.id = orders.user_id;

-- Result: All users, including those without orders
| name    | total |
|---------|-------|
| Alice   | 100   |
| Alice   | 200   |
| Bob     | 150   |
| Charlie | NULL  |
-- Charlie included with NULL for order data

-- Use when: You want all records from the first table,
-- even if there's no match in the second table
```

3. RIGHT JOIN (RIGHT OUTER JOIN) - All rows from right table, matching rows from left

```
SELECT users.name, orders.total
FROM users
RIGHT JOIN orders ON users.id = orders.user_id;

-- Result: All orders, even if user is missing (rare
case)
-- Similar to LEFT JOIN but reversed
```

```
-- Use when: You want all records from the second table
-- (Less common than LEFT JOIN)
```

4. **FULL OUTER JOIN** - All rows from both tables

```
SELECT users.name, orders.total
FROM users
FULL OUTER JOIN orders ON users.id = orders.user_id;

-- Result: All users AND all orders, with NULLs where
no match
| name    | total |
|---------|-------|
| Alice   | 100   |
| Alice   | 200   |
| Bob     | 150   |
| Charlie | NULL  |

-- Use when: You want to see all records from both
tables
-- Note: MySQL doesn't support FULL OUTER JOIN
-- (use UNION of LEFT and RIGHT JOIN instead)
```

Practical Example in Node.js:

```
// Get all users with their order count
const result = await pool.query(`
  SELECT
    users.id,
    users.name,
    COUNT(orders.id) as order_count,
```

```
    COALESCE(SUM(orders.total), 0) as total_spent
  FROM users
  LEFT JOIN orders ON users.id = orders.user_id
  GROUP BY users.id, users.name
  ORDER BY total_spent DESC
`);
```

Quick Reference:

- **INNER**: Intersection (matching records only)
- **LEFT**: All from left + matching from right
- **RIGHT**: All from right + matching from left
- **FULL**: Everything from both tables

25. What is an INDEX and why use it?

Answer:

Indexes improve query performance by creating a data structure for faster lookups.

```
-- Create index
CREATE INDEX idx_email ON users(email);

-- Unique index
CREATE UNIQUE INDEX idx_username ON users(username);

-- Composite index
CREATE INDEX idx_name ON users(first_name, last_name);
```

Trade-offs:

- Faster SELECT queries
- Slower INSERT/UPDATE/DELETE
- Requires additional storage

26. What are ACID properties?

Answer:

- **Atomicity**: All operations succeed or all fail
- **Consistency**: Data remains in valid state
- **Isolation**: Concurrent transactions don't interfere
- **Durability**: Committed data persists after system failure

27. What is a Transaction?

Answer:
A transaction is a sequence of database operations that are treated as a single unit of work. Either ALL operations succeed, or ALL are rolled back. This ensures data consistency.

Why use Transactions?
Imagine transferring money between bank accounts:

1. Subtract $100 from Account A
2. Add $100 to Account B

If the application crashes after step 1, Account A loses $100 but Account B never receives it! Transactions prevent this.

ACID Properties in Action:

- **Atomicity**: Both operations happen or neither happens
- **Consistency**: Total money remains the same
- **Isolation**: Other transactions don't see intermediate states
- **Durability**: Once committed, changes are permanent

SQL Syntax:

```
BEGIN;  -- Start transaction

-- If ANY of these fail, NONE of them take effect
UPDATE accounts SET balance = balance - 100 WHERE id =
1;
UPDATE accounts SET balance = balance + 100 WHERE id =
2;

COMMIT;  -- Save all changes
-- or ROLLBACK; -- Undo all changes
```

Node.js Implementation (PostgreSQL):

```
import { Pool } from 'pg';
const pool = new Pool();

// Method 1: Manual transaction control
async function  (fromId: number, toId: number, amount:
number) {
  const client = await pool.connect(); // Get dedicated
client for transaction

  try {
```

```
    await client.query('BEGIN'); // Start transaction

    // Deduct from sender
    const deductResult = await client.query(
      'UPDATE accounts SET balance = balance - $1 WHERE
id = $2 RETURNING balance',
      [amount, fromId]
    );

    // Check if sender has sufficient balance
    if (deductResult.rows[0].balance < 0) {
      throw new Error('Insufficient balance');
    }

    // Add to receiver
    await client.query(
      'UPDATE accounts SET balance = balance + $1 WHERE
id = $2',
      [amount, toId]
    );

    await client.query('COMMIT'); // Save changes
    console.log('Transfer successful');

  } catch (error) {
    await client.query('ROLLBACK'); // Undo all changes
    console.error('Transfer failed, rolling back:',
error);
    throw error;
  } finally {
    client.release(); // Return client to pool
  }
```

```
}

// Usage
try {
  await transferMoney(1, 2, 100);
} catch (error) {
  // Handle error
}
```

Method 2: Using transaction helper (cleaner):

```
async function withTransaction<T>(
  callback: (client: any) => Promise<T>
): Promise<T> {
  const client = await pool.connect();
  try {
    await client.query('BEGIN');
    const result = await callback(client);
    await client.query('COMMIT');
    return result;
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
}

// Usage
await withTransaction(async (client) => {
  await client.query('UPDATE accounts SET balance =
```

```
balance - $1 WHERE id = $2', [100, 1]);
  await client.query('UPDATE accounts SET balance =
balance + $1 WHERE id = $2', [100, 2]);
});
```

Common Use Cases:

- Financial transactions (transfers, payments)
- Creating related records (user + profile + preferences)
- Updating multiple tables that must stay in sync
- Any operation where partial completion would corrupt data

Important Notes:

- Always use try-catch with transactions
- Always release the client back to the pool (use finally)
- Keep transactions short to avoid locking issues
- Don't use the pool directly for transactions - get a dedicated client

28. What is SQL Injection and how to prevent it?

Answer:
SQL Injection is a security vulnerability where an attacker inserts malicious SQL code through user input, potentially gaining unauthorized access to or manipulating database data.

How SQL Injection Works:

```
// Vulnerable code
const userId = req.params.id; // User provides: "1 OR
```

```
1=1"
const query = `SELECT * FROM users WHERE id =
${userId}`;
// Becomes: SELECT * FROM users WHERE id = 1 OR 1=1
// Returns ALL users! (1=1 is always true)

// Worse: User provides: "1; DROP TABLE users; --"
// Becomes: SELECT * FROM users WHERE id = 1; DROP
TABLE users; --
// Your users table is DELETED!
```

Real-World Attack Example:

```
// Login vulnerability
const email = req.body.email;    // User:
admin@example.com
const password = req.body.password; // User: anything'
OR '1'='1

// BAD - String concatenation
const query = `
  SELECT * FROM users
  WHERE email = '${email}' AND password = '${password}'
`;
// Becomes: SELECT * FROM users WHERE email =
'admin@example.com' AND password = 'anything' OR
'1'='1'
// Logs in without valid password!
```

Prevention Methods:

1. Use Parameterized Queries (Best Practice):

```
// PostgreSQL - Uses $1, $2, $3 placeholders
const result = await pool.query(
  'SELECT * FROM users WHERE id = $1',
  [userId]  // Driver escapes this value safely
);

// MySQL - Uses ? placeholders
const [rows] = await pool.query(
  'SELECT * FROM users WHERE id = ?',
  [userId]
);

// The database driver treats [userId] as DATA, not
CODE
// So even if userId = "1 OR 1=1", it searches for
literal string "1 OR 1=1"
```

2. Use ORM (Object-Relational Mapping):

```
// Using Prisma ORM
const user = await prisma.user.findUnique({
  where: { id: userId }
});
// Prisma automatically uses parameterized queries

// Using Sequelize ORM
const user = await User.findByPk(userId);
// Sequelize also prevents SQL injection
```

3. Input Validation:

```
// Validate and sanitize input
import validator from 'validator';

const userId = req.params.id;

// Ensure it's actually a number
if (!validator.isInt(userId)) {
  return res.status(400).json({ error: 'Invalid user
ID' });
}

// Now safe to use (still use parameterized queries!)
const result = await pool.query(
  'SELECT * FROM users WHERE id = $1',
  [parseInt(userId)]
);
```

4. Principle of Least Privilege:

```
// Database user for your app should have LIMITED
permissions
// DON'T use root/admin user in your application

// Create restricted database user:
// GRANT SELECT, INSERT, UPDATE ON database.users TO
'app_user';
// DO NOT GRANT DROP, CREATE, ALTER permissions
```

Complete Safe Example:

```
import { body, validationResult } from 'express-
validator';

app.post('/login',
  // Validate input
  body('email').isEmail().normalizeEmail(),
  body('password').isLength({ min: 6 }),
  async (req, res) => {
    // Check validation
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors:
errors.array() });
    }

    const { email, password } = req.body;

    // Use parameterized query
    const result = await pool.query(
      'SELECT * FROM users WHERE email = $1',
      [email]  // Safe from SQL injection
    );

    // Always use password hashing (bcrypt)
    const user = result.rows[0];
    if (!user || !(await bcrypt.compare(password,
user.password))) {
      return res.status(401).json({ error: 'Invalid
credentials' });
    }
```

```
    res.json({ token: generateToken(user) });
  }
);
```

Never Do This:

```
// ❌ String concatenation
const query = `SELECT * FROM users WHERE id = ${id}`;

// ❌ Template literals
const query = `SELECT * FROM users WHERE email =
'${email}'`;

// ❌ String addition
const query = 'SELECT * FROM users WHERE id = ' + id;
```

Always Do This:

```
// ✅ Parameterized queries
const query = 'SELECT * FROM users WHERE id = $1';
await pool.query(query, [id]);

// ✅ ORM
await User.findByPk(id);
```

Key Takeaway: Never trust user input. Always use parameterized queries or ORMs.

29. What is the difference between `TRUNCATE` and `DELETE`?

Answer:

Feature	DELETE	TRUNCATE
WHERE clause	Yes	No
Rollback	Yes	No (in most cases)
Speed	Slower	Faster
Triggers	Fired	Not fired
Identity reset	No	Yes

```
DELETE FROM users WHERE active = false;  -- Removes
specific rows
TRUNCATE TABLE users;                    -- Removes all
rows quickly
```

30. How do you connect to PostgreSQL/MySQL in Node.js?

Answer:

PostgreSQL (pg):

```
import { Pool } from 'pg';

const pool = new Pool({
  host: 'localhost',
  port: 5432,
  database: 'mydb',
  user: 'postgres',
  password: 'password'
```

```
});

const result = await pool.query('SELECT * FROM users
WHERE id = $1', [userId]);
```

MySQL (mysql2):

```
import mysql from 'mysql2/promise';

const pool = mysql.createPool({
  host: 'localhost',
  user: 'root',
  password: 'password',
  database: 'mydb'
});

const [rows] = await pool.query('SELECT * FROM users
WHERE id = ?', [userId]);
```

TypeScript Basics

31. What is TypeScript and why use it?

Answer:

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.

Benefits:

- Static type checking

- Better IDE support
- Early error detection
- Enhanced code documentation
- Refactoring support

32. What are the basic types in TypeScript?

Answer:

```
// Primitive types
const name: string = 'John';
const age: number = 25;
const isActive: boolean = true;
const value: null = null;
const data: undefined = undefined;

// Arrays
const numbers: number[] = [1, 2, 3];
const names: Array<string> = ['John', 'Jane'];

// Tuple
const user: [string, number] = ['John', 25];

// Any (avoid when possible)
const anything: any = 'could be anything';

// Unknown (safer than any)
const userInput: unknown = getUserInput();
```

33. What is the difference between `interface` and `type`?

Answer:

```
// Interface - can be extended and merged
interface User {
  id: number;
  name: string;
}

interface User {
  email: string; // Declaration merging
}

interface Admin extends User {
  role: string;
}

// Type - more flexible, can use unions
type ID = number | string;

type User = {
  id: ID;
  name: string;
};

type Admin = User & {
  role: string;
};
```

When to use:

- Use `interface` for object shapes that might be extended

- Use `type` for unions, intersections, primitives

34. What are Union and Intersection types?

Answer:

```
// Union - can be one of several types
type Status = 'pending' | 'approved' | 'rejected';
type ID = number | string;

function processId(id: ID) {
  if (typeof id === 'string') {
    return id.toUpperCase();
  }
  return id;
}

// Intersection - combines multiple types
type Person = {
  name: string;
};

type Employee = {
  employeeId: number;
};

type Staff = Person & Employee;

const staff: Staff = {
  name: 'John',
```

```
  employeeId: 123
};
```

35. What are Generics?

Answer:

```
// Generic function
function getFirstElement<T>(arr: T[]): T | undefined {
  return arr[0];
}

const firstNumber = getFirstElement<number>([1, 2, 3]);
// number
const firstName = getFirstElement<string>(['a', 'b']);
// string

// Generic interface
interface ApiResponse<T> {
  data: T;
  status: number;
  message: string;
}

interface User {
  id: number;
  name: string;
}

const response: ApiResponse<User> = {
  data: { id: 1, name: 'John' },
```

```
  status: 200,
  message: 'Success'
};
```

36. What is `enum` in TypeScript?

Answer:

```
// Numeric enum
enum UserRole {
  Admin,      // 0
  Moderator,  // 1
  User        // 2
}

// String enum
enum Status {
  Pending = 'PENDING',
  Approved = 'APPROVED',
  Rejected = 'REJECTED'
}

function checkRole(role: UserRole) {
  if (role === UserRole.Admin) {
    // Admin access
  }
}
```

37. What is the difference between `any`, `unknown`, and `never`?

Answer:

```
// any - disables type checking
let value: any = 'hello';
value.toUpperCase(); // No error

// unknown - type-safe version of any
let userInput: unknown = getUserInput();
// userInput.toUpperCase(); // Error!
if (typeof userInput === 'string') {
  userInput.toUpperCase(); // OK
}

// never - represents values that never occur
function throwError(message: string): never {
  throw new Error(message);
}

function infiniteLoop(): never {
  while (true) {}
}
```

38. What are Optional and Default parameters?

Answer:

```
// Optional parameters
function greet(name: string, greeting?: string) {
  return `${greeting || 'Hello'}, ${name}`;
}

// Default parameters
function createUser(name: string, role: string =
```

```
'user') {
  return { name, role };
}

// Optional properties
interface User {
  id: number;
  name: string;
  email?: string; // Optional
}
```

39. What is Type Assertion?

Answer:

```
// Type assertion - telling TypeScript you know better
const input = document.getElementById('input') as
HTMLInputElement;
input.value = 'Hello';

// Angle bracket syntax (not in JSX)
const input2 =
<HTMLInputElement>document.getElementById('input');

// Const assertion
const config = {
  apiUrl: 'https://api.example.com',
  timeout: 5000
} as const; // Makes properties readonly
```

40. What are Utility Types?

Answer:

```
interface User {
  id: number;
  name: string;
  email: string;
  age: number;
}

// Partial - makes all properties optional
type PartialUser = Partial<User>;

// Required - makes all properties required
type RequiredUser = Required<User>;

// Pick - select specific properties
type UserPreview = Pick<User, 'id' | 'name'>;

// Omit - exclude specific properties
type UserWithoutId = Omit<User, 'id'>;

// Record - creates object type with keys and value
type
type UserRoles = Record<string, string>;

// Readonly - makes properties readonly
type ReadonlyUser = Readonly<User>;
```

REST APIs

41. What is REST API?

Answer:

REST (Representational State Transfer) is an architectural style for designing networked applications.

Principles:

- Client-Server architecture
- Stateless communication
- Cacheable responses
- Uniform interface
- Layered system

42. What are HTTP Status Codes?

Answer:

Common status codes:

- **200**: OK - Success
- **201**: Created - Resource created
- **204**: No Content - Success, no response body
- **400**: Bad Request - Invalid request
- **401**: Unauthorized - Authentication required
- **403**: Forbidden - No permission
- **404**: Not Found - Resource doesn't exist

- **422**: Unprocessable Entity - Validation error
- **500**: Internal Server Error - Server error
- **503**: Service Unavailable - Server down

43. What are REST API best practices?

Answer:

```
// 1. Use nouns for resources, not verbs
GET    /api/users      // Good
GET    /api/getUsers   // Bad

// 2. Use HTTP methods correctly
GET    /api/users      // Get all users
POST   /api/users      // Create user
GET    /api/users/:id  // Get one user
PUT    /api/users/:id  // Update user (full)
PATCH  /api/users/:id  // Update user (partial)
DELETE /api/users/:id  // Delete user

// 3. Use plural nouns
/api/users    // Good
/api/user     // Bad

// 4. Use query parameters for filtering
GET /api/users?role=admin&active=true

// 5. Version your API
/api/v1/users
```

44. How do you implement pagination?

Answer:

Pagination divides large datasets into smaller pages, improving performance and user experience. Instead of sending 10,000 users at once, send 10 at a time.

Why use pagination?

- **Performance**: Reduces data transfer and query time
- **User Experience**: Easier to navigate small chunks of data
- **Server Load**: Reduces memory usage and database load
- **Network**: Faster response times with smaller payloads

Common Pagination Approaches:

1. **Offset-based** (page/limit): Simple, works for most cases
2. **Cursor-based** (keyset): Better for real-time data, prevents skipping items
3. **Seek Method**: Most efficient for large datasets

Offset-Based Pagination (Most Common):

How it works:

```
Database has 100 users
Page 1: Users 1-10   (OFFSET 0, LIMIT 10)
Page 2: Users 11-20  (OFFSET 10, LIMIT 10)
Page 3: Users 21-30  (OFFSET 20, LIMIT 10)

Formula: OFFSET = (page - 1) × limit
```

Implementation:

```
interface PaginationParams {
  page: number;
  limit: number;
  offset: number;
}

interface PaginationMeta {
  page: number;
  limit: number;
  total: number;
  totalPages: number;
  hasNextPage: boolean;
  hasPrevPage: boolean;
}

interface PaginatedResponse<T> {
  data: T[];
  pagination: PaginationMeta;
}

app.get('/api/users', async (req: Request, res:
Response) => {
  try {
    // 1. Parse and validate query parameters
    const page = Math.max(1, parseInt(req.query.page as
string) || 1);
    const limit = Math.min(100, Math.max(1,
parseInt(req.query.limit as string) || 10));
    // Min 1, Max 100 per page (prevent abuse)
```

```
    const offset = (page - 1) * limit;

    // 2. Run both queries in parallel for better
performance
    const [usersResult, totalResult] = await
Promise.all([
      pool.query(
        'SELECT id, name, email, created_at FROM users
ORDER BY created_at DESC LIMIT $1 OFFSET $2',
        [limit, offset]
      ),
      pool.query('SELECT COUNT(*) FROM users')
    ]);

    const users = usersResult.rows;
    const total = parseInt(totalResult.rows[0].count);
    const totalPages = Math.ceil(total / limit);

    // 3. Build response with pagination metadata
    const response: PaginatedResponse<any> = {
      data: users,
      pagination: {
        page,
        limit,
        total,
        totalPages,
        hasNextPage: page < totalPages,
        hasPrevPage: page > 1
      }
    };

    res.json(response);
```

```
  } catch (error) {
    res.status(500).json({ error: 'Failed to fetch
users' });
  }
});

// Example response:
// {
//   "data": [
//     { "id": 1, "name": "Alice", "email":
"alice@example.com" },
//     { "id": 2, "name": "Bob", "email":
"bob@example.com" }
//   ],
//   "pagination": {
//     "page": 2,
//     "limit": 10,
//     "total": 45,
//     "totalPages": 5,
//     "hasNextPage": true,
//     "hasPrevPage": true
//   }
// }
```

Client-Side Usage:

```
// Initial request
fetch('/api/users?page=1&limit=10')
  .then(res => res.json())
  .then(data => {
    console.log(data.data); // Array of users
```

```
    console.log(data.pagination); // Pagination info
  });

// Next page
fetch('/api/users?page=2&limit=10');

// Different page size
fetch('/api/users?page=1&limit=25');
```

Advanced: Pagination with Filtering and Sorting:

```
app.get('/api/users', async (req: Request, res:
Response) => {
  const page = Math.max(1, parseInt(req.query.page as
string) || 1);
  const limit = Math.min(100, parseInt(req.query.limit
as string) || 10);
  const offset = (page - 1) * limit;

  // Filters
  const role = req.query.role as string;
  const search = req.query.search as string;

  // Sorting
  const sortBy = req.query.sortBy as string ||
'created_at';
  const order = req.query.order === 'asc' ? 'ASC' :
'DESC';

  // Build dynamic query
  let whereClause = 'WHERE 1=1';
```

```
  const params: any[] = [];
  let paramCount = 1;

  if (role) {
    whereClause += ` AND role = $${paramCount++}`;
    params.push(role);
  }

  if (search) {
    whereClause += ` AND (name ILIKE $${paramCount++}
OR email ILIKE $${paramCount})`;
    params.push(`%${search}%`, `%${search}%`);
    paramCount++;
  }

  // Validate sortBy to prevent SQL injection
  const validSortFields = ['name', 'email',
'created_at', 'id'];
  const sortField = validSortFields.includes(sortBy) ?
sortBy : 'created_at';

  const query = `
    SELECT id, name, email, role, created_at
    FROM users
    ${whereClause}
    ORDER BY ${sortField} ${order}
    LIMIT $${paramCount++} OFFSET $${paramCount}
  `;

  params.push(limit, offset);

  const countQuery = `SELECT COUNT(*) FROM users
```

```
${whereClause}`;

  const [usersResult, totalResult] = await
Promise.all([
    pool.query(query, params),
    pool.query(countQuery, params.slice(0,
params.length - 2)) // Exclude limit/offset
  ]);

  const total = parseInt(totalResult.rows[0].count);

  res.json({
    data: usersResult.rows,
    pagination: {
      page,
      limit,
      total,
      totalPages: Math.ceil(total / limit),
      hasNextPage: page < Math.ceil(total / limit),
      hasPrevPage: page > 1
    },
    filters: {
      role,
      search,
      sortBy: sortField,
      order
    }
  });
});

// Usage: /api/users?
```

```
page=1&limit=10&role=admin&search=john&sortBy=name&orde
r=asc
```

Cursor-Based Pagination (Better for real-time data):

```
// Uses ID as cursor instead of page number
app.get('/api/posts', async (req: Request, res:
Response) => {
  const limit = Math.min(100, parseInt(req.query.limit
as string) || 20);
  const cursor = req.query.cursor as string; // Last ID
from previous page

  let query;
  let params;

  if (cursor) {
    // Get posts after cursor
    query = 'SELECT * FROM posts WHERE id > $1 ORDER BY
id ASC LIMIT $2';
    params = [cursor, limit + 1]; // +1 to check if
there's a next page
  } else {
    // First page
    query = 'SELECT * FROM posts ORDER BY id ASC LIMIT
$1';
    params = [limit + 1];
  }

  const result = await pool.query(query, params);
  const posts = result.rows;
```

```
  const hasNextPage = posts.length > limit;
  if (hasNextPage) {
    posts.pop(); // Remove extra item
  }

  const nextCursor = hasNextPage ? posts[posts.length -
1].id : null;

  res.json({
    data: posts,
    pagination: {
      nextCursor,
      hasNextPage,
      limit
    }
  });
});

// Usage:
// Page 1: /api/posts?limit=20
// Page 2: /api/posts?limit=20&cursor=20
// Page 3: /api/posts?limit=20&cursor=40
```

Reusable Pagination Helper:

```
class PaginationHelper {
  static paginate<T>(
    items: T[],
    total: number,
    page: number,
```

```
    limit: number
  ): PaginatedResponse<T> {
    const totalPages = Math.ceil(total / limit);

    return {
      data: items,
      pagination: {
        page,
        limit,
        total,
        totalPages,
        hasNextPage: page < totalPages,
        hasPrevPage: page > 1,
        nextPage: page < totalPages ? page + 1 : null,
        prevPage: page > 1 ? page - 1 : null
      }
    };
  }

  static getPaginationParams(query: any) {
    const page = Math.max(1, parseInt(query.page) ||
1);
    const limit = Math.min(100, Math.max(1,
parseInt(query.limit) || 10));
    const offset = (page - 1) * limit;

    return { page, limit, offset };
  }
}

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

```
  const { page, limit, offset } =
PaginationHelper.getPaginationParams(req.query);

  const [users, total] = await Promise.all([
    getUsers(limit, offset),
    getUsersCount()
  ]);

  res.json(PaginationHelper.paginate(users, total,
page, limit));
});
```

Performance Tips:

```
// 1. Index the column used for sorting
CREATE INDEX idx_users_created_at ON users(created_at);

// 2. Avoid COUNT(*) on large tables (expensive)
// Cache the count or use approximate count
// Redis cache: await redis.get('users:count')

// 3. Use cursor pagination for infinite scroll
// Better performance on large datasets

// 4. Set reasonable max limit
const MAX_LIMIT = 100;
const limit = Math.min(MAX_LIMIT, requestedLimit);
```

Common Pitfalls:

- ❌ No validation on page/limit (allows page=999999999)

- ❌ Not handling page > totalPages
- ❌ Expensive COUNT(*) on every request
- ❌ No maximum limit (user requests 1 million items)
- ❌ Offset pagination on frequently changing data (items may be skipped/duplicated)

45. How do you implement filtering and sorting?

Answer:

```
app.get('/api/users', async (req: Request, res:
Response) => {
  const { role, active, sortBy, order } = req.query;

  let query = 'SELECT * FROM users WHERE 1=1';
  const params: any[] = [];
  let paramCount = 1;

  if (role) {
    query += ` AND role = $${paramCount++}`;
    params.push(role);
  }

  if (active !== undefined) {
    query += ` AND active = $${paramCount++}`;
    params.push(active === 'true');
  }

  if (sortBy) {
    const validSortFields = ['name', 'created_at',
```

```
'email'];
    const sortField = validSortFields.includes(sortBy
as string) ? sortBy : 'created_at';
    const sortOrder = order === 'desc' ? 'DESC' :
'ASC';
    query += ` ORDER BY ${sortField} ${sortOrder}`;
  }

  const result = await pool.query(query, params);
  res.json(result.rows);
});
```

Authentication & Security

46. What is JWT (JSON Web Token)?

Answer:

JWT is a compact, self-contained token format for securely transmitting information between parties. It's commonly used for authentication and information exchange.

Why use JWT?

- **Stateless**: Server doesn't need to store sessions (scales better)
- **Self-contained**: Token contains all necessary user information
- **Cross-domain**: Works across different domains and services
- **Mobile-friendly**: Perfect for mobile apps (no cookies needed)

JWT Structure (3 parts separated by dots):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEsIm
VtYWlsIjoidGVzdEB0ZXN0LmNvbSJ9.Xjs1P2Q3R4S5T6U7V8W9X
│                                          │
│
└─ Header (algorithm & type)              └─ Payload
(claims/data)                                 └─
Signature (verification)
```

1. Header: Algorithm and token type

```
{
  "alg": "HS256",    // Algorithm: HMAC SHA-256
  "typ": "JWT"       // Type: JWT
}
```

2. Payload: Claims (user data)

```
{
  "userId": 1,
  "email": "test@example.com",
  "role": "user",
  "iat": 1516239022,    // Issued At
  "exp": 1516242622     // Expiration Time
}
```

3. Signature: Verification (prevents tampering)

```
HMACSHA256(
  base64UrlEncode(header) + "." +
base64UrlEncode(payload),
```

```
  secret
)
```

How JWT Authentication Works:

```
1. Client sends credentials (email/password)
   ↓
2. Server verifies credentials
   ↓
3. Server creates JWT with user info
   ↓
4. Server sends JWT to client
   ↓
5. Client stores JWT (localStorage/cookie)
   ↓
6. Client sends JWT with each request (Authorization
header)
   ↓
7. Server verifies JWT signature
   ↓
8. Server grants access if valid
```

Implementation:

```
import jwt from 'jsonwebtoken';

// Configuration
const JWT_SECRET = process.env.JWT_SECRET!; // Keep
this secret!
const JWT_EXPIRES_IN = '24h'; // Token lifetime
```

```
// 1. Generate token (after successful login)
function generateToken(user: { id: number; email:
string; role: string }) {
  const payload = {
    userId: user.id,
    email: user.email,
    role: user.role
  };

  const token = jwt.sign(
    payload,                  // Data to encode
    JWT_SECRET,               // Secret key (never
expose this!)
    {
      expiresIn: JWT_EXPIRES_IN,  // Token expires in
24 hours
      issuer: 'my-app',           // Optional: who
issued it
      audience: 'my-app-users'    // Optional: who can
use it
    }
  );

  return token;
}

// 2. Verify token (on protected routes)
function verifyToken(token: string) {
  try {
    const decoded = jwt.verify(token, JWT_SECRET);
    // decoded = { userId: 1, email:
'test@example.com', role: 'user', iat: ..., exp: ... }
```

```
    return decoded;
  } catch (error) {
    if (error instanceof jwt.TokenExpiredError) {
      throw new Error('Token expired');
    }
    if (error instanceof jwt.JsonWebTokenError) {
      throw new Error('Invalid token');
    }
    throw error;
  }
}

// 3. Login endpoint
app.post('/api/login', async (req, res) => {
  const { email, password } = req.body;

  // Verify credentials (check database)
  const user = await findUserByEmail(email);
  if (!user || !(await bcrypt.compare(password,
user.password))) {
    return res.status(401).json({ error: 'Invalid
credentials' });
  }

  // Generate token
  const token = generateToken({
    id: user.id,
    email: user.email,
    role: user.role
  });

  // Send token to client
```

```
  res.json({
    message: 'Login successful',
    token,
    user: {
      id: user.id,
      email: user.email,
      name: user.name
    }
  });
});

// 4. Client stores and uses token
// Client-side (React/Angular/Vue):
// localStorage.setItem('token', token);
//
// Then sends with each request:
// headers: { 'Authorization': `Bearer ${token}` }
```

Security Best Practices:

```
// 1. Use strong secret (256-bit minimum)
const JWT_SECRET =
crypto.randomBytes(64).toString('hex');

// 2. Set reasonable expiration
const shortLived = '15m';   // For sensitive operations
const normal = '1h';        // For regular access
const longLived = '7d';     // For "remember me"

// 3. Use HTTPS only (prevent token interception)
```

```
// 4. Store tokens securely on client
// Good: HttpOnly cookie (prevents XSS)
// OK: localStorage (convenient but vulnerable to XSS)
// Bad: Regular cookie without HttpOnly

// 5. Implement token refresh mechanism
function generateRefreshToken(userId: number) {
  return jwt.sign(
    { userId, type: 'refresh' },
    REFRESH_TOKEN_SECRET,
    { expiresIn: '7d' }
  );
}

// 6. Include token ID for revocation
const payload = {
  userId: user.id,
  jti: uuid(), // JWT ID - can track/revoke specific
tokens
  email: user.email
};
```

Common Pitfalls:

- ❌ Storing sensitive data in payload (it's Base64, not encrypted!)
- ❌ Not setting expiration time
- ❌ Using weak secrets
- ❌ Not validating token on every protected route
- ❌ Storing tokens in plain cookies (use HttpOnly flag)

JWT vs Sessions:

Feature	JWT	Session
Storage	Client-side	Server-side
Scalability	Better (stateless)	Requires shared storage
Revocation	Difficult	Easy
Size	Larger	Smaller
Best for	Microservices, APIs	Traditional web apps

47. How do you implement authentication middleware?

Answer:

Authentication middleware verifies that a user is logged in by checking their JWT token. It runs before protected routes to ensure only authenticated users can access them.

How it works:

```
1. Client sends request with JWT in Authorization
header
   Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
   ↓
2. Middleware extracts token from header
   ↓
3. Middleware verifies token signature and expiration
```

```
   ↓
4. If valid: Attach user info to request, call next()
   If invalid: Send 401 Unauthorized response
   ↓
5. Route handler receives request with user info
```

Implementation:

```
import { Request, Response, NextFunction } from
'express';
import jwt from 'jsonwebtoken';

// Extend Request type to include user property
interface AuthRequest extends Request {
  user?: {
    userId: number;
    email: string;
    role: string;
  };
}

// Authentication middleware
export const authMiddleware = (
  req: AuthRequest,
  res: Response,
  next: NextFunction
) => {
  // 1. Extract token from Authorization header
  // Expected format: "Bearer <token>"
  const authHeader = req.headers.authorization;
```

```
  if (!authHeader) {
    return res.status(401).json({ error: 'No
authorization header' });
  }

  // Split "Bearer token" and get token part
  const token = authHeader.split(' ')[1];

  if (!token) {
    return res.status(401).json({ error: 'No token
provided' });
  }

  try {
    // 2. Verify token (checks signature and
expiration)
    const decoded = jwt.verify(
      token,
      process.env.JWT_SECRET!
    ) as {
      userId: number;
      email: string;
      role: string;
    };

    // 3. Attach user info to request object
    // Now available in all subsequent
middleware/handlers
    req.user = decoded;

    // 4. Continue to next middleware/route handler
    next();
```

```
  } catch (error) {
    // Handle different error types
    if (error instanceof jwt.TokenExpiredError) {
      return res.status(401).json({
        error: 'Token expired',
        code: 'TOKEN_EXPIRED'
      });
    }

    if (error instanceof jwt.JsonWebTokenError) {
      return res.status(401).json({
        error: 'Invalid token',
        code: 'INVALID_TOKEN'
      });
    }

    // Unexpected error
    return res.status(500).json({ error:
'Authentication error' });
  }
};

// Optional: Middleware to check user roles
export const authorize = (...allowedRoles: string[]) =>
{
  return (req: AuthRequest, res: Response, next:
NextFunction) => {
    // First check if user is authenticated
    if (!req.user) {
      return res.status(401).json({ error: 'Not
authenticated' });
```

```
    }

    // Then check if user has required role
    if (!allowedRoles.includes(req.user.role)) {
      return res.status(403).json({
        error: 'Forbidden',
        message: `Requires one of:
${allowedRoles.join(', ')}`
      });
    }

    next();
  };
};

// Usage Examples:

// 1. Protect single route
app.get('/api/profile', authMiddleware, (req:
AuthRequest, res) => {
  // req.user is now available here
  res.json({
    message: 'This is your profile',
    user: req.user
  });
});

// 2. Protect all routes under /api
app.use('/api', authMiddleware);

// 3. Combine authentication + authorization
app.delete(
```

```
  '/api/users/:id',
  authMiddleware,           // Must be logged in
  authorize('admin'),       // Must be admin role
  async (req: AuthRequest, res) => {
    // Only admins can delete users
    await deleteUser(req.params.id);
    res.json({ message: 'User deleted' });
  }
);

// 4. Multiple role authorization
app.post(
  '/api/posts',
  authMiddleware,
  authorize('admin', 'moderator', 'author'), // Any of
these roles
  async (req: AuthRequest, res) => {
    // Create post
  }
);

// 5. Optional authentication (public route but adds
user if logged in)
const optionalAuth = (req: AuthRequest, res: Response,
next: NextFunction) => {
  const token = req.headers.authorization?.split(' ')
[1];

  if (token) {
    try {
      req.user = jwt.verify(token,
process.env.JWT_SECRET!) as any;
```

```
    } catch {
      // Token invalid but don't block request
    }
  }

  next();
};

app.get('/api/posts', optionalAuth, async (req:
AuthRequest, res) => {
  // Public route, but can check if user is logged in
  const posts = await getPosts();

  if (req.user) {
    // Logged in - include user-specific data
    return res.json({ posts, favorites: await
getUserFavorites(req.user.userId) });
  }

  // Not logged in - just posts
  res.json({ posts });
});
```

Client-Side Usage:

```
// Store token after login
localStorage.setItem('token', token);

// Send token with each request
fetch('/api/profile', {
  headers: {
```

```
    'Authorization': `Bearer
${localStorage.getItem('token')}`
  }
});

// Using Axios
axios.defaults.headers.common['Authorization'] =
`Bearer ${token}`;

// Or using interceptor
axios.interceptors.request.use(config => {
  const token = localStorage.getItem('token');
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});
```

Testing Authentication:

```
// In your tests
const token = jwt.sign({ userId: 1, role: 'admin' },
JWT_SECRET);

const response = await request(app)
  .get('/api/profile')
  .set('Authorization', `Bearer ${token}`)
  .expect(200);
```

48. How do you hash passwords?

Answer:

Password hashing is a one-way cryptographic function that converts a password into a fixed-length string. You should **NEVER** store plain-text passwords in a database!

Why hash passwords?

- If database is compromised, hackers can't see actual passwords
- Users often reuse passwords across sites - hashing protects them
- It's a legal/compliance requirement (GDPR, PCI-DSS)

How bcrypt works:

1. **Salt**: Random data added to password before hashing (prevents rainbow table attacks)
2. **Cost Factor**: How many iterations to run (higher = slower = more secure)
3. **One-way**: Cannot reverse a hash back to the original password
4. **Adaptive**: Can increase cost factor as computers get faster

bcrypt vs other algorithms:

- ❌ **MD5/SHA-1**: Fast, insecure (easily cracked)
- ⚠️ **SHA-256**: Fast, not ideal for passwords (no salt, too fast)
- ✅ **bcrypt**: Slow by design, has built-in salt, industry standard
- ✅ **argon2**: Modern alternative, winner of password hashing competition
- ✅ **scrypt**: Also good, more memory-intensive

Implementation:

```
import bcrypt from 'bcrypt';

// Configuration
const SALT_ROUNDS = 10; // Higher = more secure but
slower
// 10 = ~100ms to hash (good balance)
// 12 = ~300ms to hash (more secure)
// 15 = ~3 seconds to hash (very secure but slow)

// 1. Hash password (during registration)
const hashPassword = async (password: string):
Promise<string> => {
  // bcrypt.hash automatically generates a salt and
includes it in the hash
  const hash = await bcrypt.hash(password,
SALT_ROUNDS);
  // Returns: $2b$10$N9qo8uLOickgx2ZMRZoMye... (60
characters)
  // Format: $2b$ <cost> $ <salt> <hash>
  return hash;
};

// 2. Verify password (during login)
const comparePassword = async (
  plainPassword: string,
  hashedPassword: string
): Promise<boolean> => {
  // bcrypt extracts salt from hashedPassword and
compares
  // Returns true if match, false if not
  return await bcrypt.compare(plainPassword,
```

```
hashedPassword);
};

// 3. Registration endpoint
app.post('/api/register', async (req, res) => {
  try {
    const { email, password, name } = req.body;

    // Validate password strength
    if (password.length < 8) {
      return res.status(400).json({
        error: 'Password must be at least 8 characters'
      });
    }

    // Check if user already exists
    const existing = await pool.query(
      'SELECT id FROM users WHERE email = $1',
      [email]
    );

    if (existing.rows.length > 0) {
      return res.status(400).json({ error: 'Email
already registered' });
    }

    // Hash password (this takes ~100ms)
    const hashedPassword = await
hashPassword(password);

    // Store hashed password in database
    const result = await pool.query(
```

```
      'INSERT INTO users (email, password, name) VALUES
($1, $2, $3) RETURNING id, email, name',
      [email, hashedPassword, name]
    );

    const user = result.rows[0];

    // Generate JWT token
    const token = jwt.sign(
      { userId: user.id, email: user.email },
      process.env.JWT_SECRET!,
      { expiresIn: '24h' }
    );

    res.status(201).json({
      message: 'User created successfully',
      user: {
        id: user.id,
        email: user.email,
        name: user.name
      },
      token
    });
  } catch (error) {
    res.status(500).json({ error: 'Registration failed'
});
  }
});

// 4. Login endpoint
app.post('/api/login', async (req, res) => {
  try {
```

```
    const { email, password } = req.body;

    if (!email || !password) {
      return res.status(400).json({ error: 'Email and
password required' });
    }

    // Get user from database
    const result = await pool.query(
      'SELECT id, email, name, password FROM users
WHERE email = $1',
      [email]
    );

    const user = result.rows[0];

    // Check if user exists and password matches
    // Use single error message for both cases
(security: don't reveal if email exists)
    if (!user || !(await comparePassword(password,
user.password))) {
      return res.status(401).json({ error: 'Invalid
email or password' });
    }

    // Generate token
    const token = jwt.sign(
      { userId: user.id, email: user.email },
      process.env.JWT_SECRET!,
      { expiresIn: '24h' }
    );
```

```
    res.json({
      message: 'Login successful',
      user: {
        id: user.id,
        email: user.email,
        name: user.name
        // Never send password, even hashed!
      },
      token
    });
  } catch (error) {
    res.status(500).json({ error: 'Login failed' });
  }
});
```

Understanding the hash:

```
const password = 'myPassword123';
const hash = await bcrypt.hash(password, 10);

console.log(hash);
// $2b$10$N9qo8uLOickgx2ZMRZoMye...
// |  |  |                     |
// |  |  └─ Salt (22 chars)      └─ Hash (31 chars)
// |  └─ Cost factor (10)
// └─ Algorithm version (2b)

// Same password hashed twice produces DIFFERENT hashes
(due to random salt)
const hash1 = await bcrypt.hash('password', 10);
const hash2 = await bcrypt.hash('password', 10);
```

```
console.log(hash1 === hash2); // false

// But bcrypt.compare still works!
await bcrypt.compare('password', hash1); // true
await bcrypt.compare('password', hash2); // true
```

Additional Security Measures:

```
// 1. Password strength validation
import validator from 'validator';

function validatePasswordStrength(password: string):
string[] {
  const errors: string[] = [];

  if (password.length < 8) {
    errors.push('Password must be at least 8
characters');
  }

  if (!/[A-Z]/.test(password)) {
    errors.push('Password must contain uppercase
letter');
  }

  if (!/[a-z]/.test(password)) {
    errors.push('Password must contain lowercase
letter');
  }

  if (!/[0-9]/.test(password)) {
```

```
    errors.push('Password must contain a number');
  }

  if (!/[!@#$%^&*]/.test(password)) {
    errors.push('Password must contain special
character');
  }

  return errors;
}

// 2. Rate limiting (prevent brute force attacks)
import rateLimit from 'express-rate-limit';

const loginLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 5, // 5 attempts
  message: 'Too many login attempts, please try again
later'
});

app.post('/api/login', loginLimiter, async (req, res)
=> {
  // Login logic
});

// 3. Account lockout after failed attempts
let failedAttempts = {}; // In production, use Redis

app.post('/api/login', async (req, res) => {
  const { email, password } = req.body;
```

```
  // Check if account is locked
  if (failedAttempts[email]?.count >= 5) {
    const lockTime = failedAttempts[email].lockedUntil;
    if (Date.now() < lockTime) {
      return res.status(429).json({
        error: 'Account locked. Try again later.'
      });
    } else {
      delete failedAttempts[email]; // Unlock
    }
  }

  const user = await findUserByEmail(email);
  const isValid = user && await
comparePassword(password, user.password);

  if (!isValid) {
    // Track failed attempt
    failedAttempts[email] = {
      count: (failedAttempts[email]?.count || 0) + 1,
      lockedUntil: Date.now() + (15 * 60 * 1000) //
Lock for 15 min
    };
    return res.status(401).json({ error: 'Invalid
credentials' });
  }

  // Success - clear failed attempts
  delete failedAttempts[email];
  // Generate token...
});
```

Common Mistakes to Avoid:

- ❌ Storing passwords in plain text
- ❌ Using weak hashing algorithms (MD5, SHA-1)
- ❌ Not using a salt
- ❌ Rolling your own crypto
- ❌ Sending passwords in URLs or logs
- ❌ Displaying different error messages for "user not found" vs "wrong password" (security risk)

49. What is the difference between Authentication and Authorization?

Answer:

- **Authentication**: Verifying who you are (login)
- **Authorization**: Verifying what you can access (permissions)

```
// Authorization middleware
export const authorize = (...roles: string[]) => {
  return (req: AuthRequest, res: Response, next:
NextFunction) => {
    if (!req.user) {
      return res.status(401).json({ error: 'Not
authenticated' });
    }

    if (!roles.includes(req.user.role)) {
      return res.status(403).json({ error: 'Forbidden'
```

```
});
    }

    next();
  };
};

// Usage
app.delete(
  '/api/users/:id',
  authMiddleware,
  authorize('admin'),
  (req, res) => {
    // Only admins can access
  }
);
```

50. What security measures should you implement?

Answer:

```
import helmet from 'helmet';
import rateLimit from 'express-rate-limit';
import mongoSanitize from 'express-mongo-sanitize';

// 1. Helmet - Set security headers
app.use(helmet());

// 2. Rate limiting
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
```

```
  max: 100 // limit each IP to 100 requests per
windowMs
});
app.use('/api/', limiter);

// 3. Sanitize user input
app.use(mongoSanitize());

// 4. Validate input
import { body, validationResult } from 'express-
validator';

app.post('/api/users',
  body('email').isEmail(),
  body('password').isLength({ min: 6 }),
  (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors:
errors.array() });
    }
    // Process request
  }
);

// 5. Environment variables
// Never commit secrets to git
// Use .env file with dotenv package
```

Error Handling

51. How do you handle errors in Express?

Answer:

```
// Custom error class
class AppError extends Error {
  statusCode: number;
  isOperational: boolean;

  constructor(message: string, statusCode: number) {
    super(message);
    this.statusCode = statusCode;
    this.isOperational = true;
    Error.captureStackTrace(this, this.constructor);
  }
}

// Error handling middleware (must be last)
app.use((err: AppError, req: Request, res: Response,
next: NextFunction) => {
  const statusCode = err.statusCode || 500;
  const message = err.message || 'Internal Server
Error';

  console.error('Error:', err);

  res.status(statusCode).json({
    status: 'error',
    statusCode,
    message,
    ...(process.env.NODE_ENV === 'development' && {
stack: err.stack })
```

```
  });
});

// Usage
app.get('/api/users/:id', async (req, res, next) => {
  try {
    const user = await findUser(req.params.id);
    if (!user) {
      throw new AppError('User not found', 404);
    }
    res.json(user);
  } catch (error) {
    next(error);
  }
});
```

52. What is `try-catch` and `async/await`?

Answer:

```
// Without async/await (callback hell)
function getUser(id, callback) {
  db.query('SELECT * FROM users WHERE id = ?', [id],
(err, user) => {
    if (err) return callback(err);
    db.query('SELECT * FROM posts WHERE user_id = ?',
[id], (err, posts) => {
      if (err) return callback(err);
      callback(null, { user, posts });
    });
  });
```

```
}

// With async/await
const getUser = async (id: number) => {
  try {
    const user = await db.query('SELECT * FROM users
WHERE id = $1', [id]);
    const posts = await db.query('SELECT * FROM posts
WHERE user_id = $1', [id]);
    return { user: user.rows[0], posts: posts.rows };
  } catch (error) {
    throw new AppError('Error fetching user', 500);
  }
};
```

53. How do you handle async errors in Express?

Answer:

```
// Wrapper function for async routes
const asyncHandler = (fn: Function) => {
  return (req: Request, res: Response, next:
NextFunction) => {
    Promise.resolve(fn(req, res, next)).catch(next);
  };
};

// Usage
app.get('/api/users/:id', asyncHandler(async (req:
Request, res: Response) => {
  const user = await findUser(req.params.id);
```

```
  if (!user) {
    throw new AppError('User not found', 404);
  }
  res.json(user);
}));

// Or use express-async-errors package
import 'express-async-errors';

// Then just use async functions directly
app.get('/api/users/:id', async (req, res) => {
  const user = await findUser(req.params.id);
  if (!user) {
    throw new AppError('User not found', 404);
  }
  res.json(user);
});
```

Testing

54. What testing frameworks do you use for Node.js?

Answer:

- **Jest**: Popular, all-in-one testing framework
- **Mocha**: Flexible test framework
- **Chai**: Assertion library
- **Supertest**: HTTP assertion library

- **Sinon**: Mocking and stubbing

55. How do you write unit tests?

Answer:

```
// userService.ts
export class UserService {
  constructor(private db: any) {}

  async getUserById(id: number) {
    const result = await this.db.query('SELECT * FROM
users WHERE id = $1', [id]);
    return result.rows[0];
  }

  async createUser(email: string, password: string) {
    const result = await this.db.query(
      'INSERT INTO users (email, password) VALUES ($1,
$2) RETURNING *',
      [email, password]
    );
    return result.rows[0];
  }
}

// userService.test.ts
import { UserService } from './userService';

describe('UserService', () => {
  let userService: UserService;
```

```
  let mockDb: any;

  beforeEach(() => {
    mockDb = {
      query: jest.fn()
    };
    userService = new UserService(mockDb);
  });

  describe('getUserById', () => {
    it('should return user when found', async () => {
      const mockUser = { id: 1, email:
'test@example.com' };
      mockDb.query.mockResolvedValue({ rows: [mockUser]
});

      const result = await userService.getUserById(1);

      expect(result).toEqual(mockUser);
      expect(mockDb.query).toHaveBeenCalledWith(
        'SELECT * FROM users WHERE id = $1',
        [1]
      );
    });

    it('should return undefined when user not found',
async () => {
      mockDb.query.mockResolvedValue({ rows: [] });

      const result = await
userService.getUserById(999);
```

```
      expect(result).toBeUndefined();
    });
  });
});
```

56. How do you test Express routes?

Answer:

```
import request from 'supertest';
import express from 'express';
import userRoutes from './routes/users';

const app = express();
app.use(express.json());
app.use('/api/users', userRoutes);

describe('User Routes', () => {
  describe('GET /api/users', () => {
    it('should return all users', async () => {
      const response = await request(app)
        .get('/api/users')
        .expect(200);

      expect(response.body).toBeInstanceOf(Array);
    });
  });

  describe('POST /api/users', () => {
    it('should create a new user', async () => {
      const newUser = {
```

```
        email: 'test@example.com',
        password: 'password123'
      };

      const response = await request(app)
        .post('/api/users')
        .send(newUser)
        .expect(201);

      expect(response.body).toHaveProperty('id');
      expect(response.body.email).toBe(newUser.email);
    });

    it('should return 400 for invalid data', async ()
=> {
      const response = await request(app)
        .post('/api/users')
        .send({ email: 'invalid' })
        .expect(400);

      expect(response.body).toHaveProperty('error');
    });
  });
});
```

Practical Coding Questions

57. Create a simple CRUD API for Users

Answer:

```
import express, { Request, Response } from 'express';
import { Pool } from 'pg';

const app = express();
const pool = new Pool({
  connectionString: process.env.DATABASE_URL
});

app.use(express.json());

interface User {
  id?: number;
  email: string;
  name: string;
  created_at?: Date;
}

// GET all users
app.get('/api/users', async (req: Request, res:
Response) => {
  try {
    const result = await pool.query('SELECT * FROM
users ORDER BY id');
    res.json(result.rows);
  } catch (error) {
    res.status(500).json({ error: 'Server error' });
  }
});

// GET user by ID
app.get('/api/users/:id', async (req: Request, res:
```

```
Response) => {
  try {
    const { id } = req.params;
    const result = await pool.query('SELECT * FROM
users WHERE id = $1', [id]);

    if (result.rows.length === 0) {
      return res.status(404).json({ error: 'User not
found' });
    }

    res.json(result.rows[0]);
  } catch (error) {
    res.status(500).json({ error: 'Server error' });
  }
});

// POST create user
app.post('/api/users', async (req: Request, res:
Response) => {
  try {
    const { email, name }: User = req.body;

    if (!email || !name) {
      return res.status(400).json({ error: 'Email and
name are required' });
    }

    const result = await pool.query(
      'INSERT INTO users (email, name) VALUES ($1, $2)
RETURNING *',
      [email, name]
```

```
    );

    res.status(201).json(result.rows[0]);
  } catch (error) {
    res.status(500).json({ error: 'Server error' });
  }
});

// PUT update user
app.put('/api/users/:id', async (req: Request, res:
Response) => {
  try {
    const { id } = req.params;
    const { email, name }: User = req.body;

    const result = await pool.query(
      'UPDATE users SET email = $1, name = $2 WHERE id
= $3 RETURNING *',
      [email, name, id]
    );

    if (result.rows.length === 0) {
      return res.status(404).json({ error: 'User not
found' });
    }

    res.json(result.rows[0]);
  } catch (error) {
    res.status(500).json({ error: 'Server error' });
  }
});
```

```
// DELETE user
app.delete('/api/users/:id', async (req: Request, res:
Response) => {
  try {
    const { id } = req.params;
    const result = await pool.query('DELETE FROM users
WHERE id = $1 RETURNING *', [id]);

    if (result.rows.length === 0) {
      return res.status(404).json({ error: 'User not
found' });
    }

    res.json({ message: 'User deleted', user:
result.rows[0] });
  } catch (error) {
    res.status(500).json({ error: 'Server error' });
  }
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

58. Implement user registration and login

Answer:

```
import express from 'express';
import bcrypt from 'bcrypt';
```

```
import jwt from 'jsonwebtoken';
import { Pool } from 'pg';

const app = express();
const pool = new Pool({ connectionString:
process.env.DATABASE_URL });
app.use(express.json());

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

    // Validation
    if (!email || !password || !name) {
      return res.status(400).json({ error: 'All fields
required' });
    }

    if (password.length < 6) {
      return res.status(400).json({ error: 'Password
must be at least 6 characters' });
    }

    // Check if user exists
    const existingUser = await pool.query(
      'SELECT * FROM users WHERE email = $1',
      [email]
    );

    if (existingUser.rows.length > 0) {
      return res.status(400).json({ error: 'Email
```

```
already registered' });
    }

    // Hash password
    const hashedPassword = await bcrypt.hash(password,
10);

    // Create user
    const result = await pool.query(
      'INSERT INTO users (email, password, name) VALUES
($1, $2, $3) RETURNING id, email, name',
      [email, hashedPassword, name]
    );

    const user = result.rows[0];

    // Generate token
    const token = jwt.sign(
      { userId: user.id, email: user.email },
      process.env.JWT_SECRET!,
      { expiresIn: '24h' }
    );

    res.status(201).json({ user, token });
  } catch (error) {
    res.status(500).json({ error: 'Server error' });
  }
});

// Login
app.post('/api/login', async (req, res) => {
  try {
```

```
    const { email, password } = req.body;

    // Validation
    if (!email || !password) {
      return res.status(400).json({ error: 'Email and
password required' });
    }

    // Find user
    const result = await pool.query(
      'SELECT * FROM users WHERE email = $1',
      [email]
    );

    const user = result.rows[0];

    if (!user) {
      return res.status(401).json({ error: 'Invalid
credentials' });
    }

    // Check password
    const isValidPassword = await
bcrypt.compare(password, user.password);

    if (!isValidPassword) {
      return res.status(401).json({ error: 'Invalid
credentials' });
    }

    // Generate token
    const token = jwt.sign(
```

```
      { userId: user.id, email: user.email },
      process.env.JWT_SECRET!,
      { expiresIn: '24h' }
    );

    res.json({
      user: {
        id: user.id,
        email: user.email,
        name: user.name
      },
      token
    });
  } catch (error) {
    res.status(500).json({ error: 'Server error' });
  }
});
```

59. Create a middleware to validate request body

Answer:

```
import { Request, Response, NextFunction } from
'express';

interface ValidationRule {
  field: string;
  type: 'string' | 'number' | 'boolean' | 'email';
  required?: boolean;
  minLength?: number;
  maxLength?: number;
```

```
  min?: number;
  max?: number;
}

const validate = (rules: ValidationRule[]) => {
  return (req: Request, res: Response, next:
NextFunction) => {
    const errors: string[] = [];

    for (const rule of rules) {
      const value = req.body[rule.field];

      // Check required
      if (rule.required && (value === undefined ||
value === null || value === '')) {
        errors.push(`${rule.field} is required`);
        continue;
      }

      // Skip validation if not required and empty
      if (!rule.required && !value) {
        continue;
      }

      // Type validation
      if (rule.type === 'string' && typeof value !==
'string') {
        errors.push(`${rule.field} must be a string`);
      }

      if (rule.type === 'number' && typeof value !==
'number') {
```

```
        errors.push(`${rule.field} must be a number`);
      }

      if (rule.type === 'email' && typeof value ===
'string') {
        const emailRegex = /^[^\s@]+@[^\s@]+\.
[^\s@]+$/;
        if (!emailRegex.test(value)) {
          errors.push(`${rule.field} must be a valid
email`);
        }
      }

      // String length validation
      if (rule.type === 'string' && typeof value ===
'string') {
        if (rule.minLength && value.length <
rule.minLength) {
          errors.push(`${rule.field} must be at least
${rule.minLength} characters`);
        }
        if (rule.maxLength && value.length >
rule.maxLength) {
          errors.push(`${rule.field} must be at most
${rule.maxLength} characters`);
        }
      }

      // Number range validation
      if (rule.type === 'number' && typeof value ===
'number') {
        if (rule.min !== undefined && value < rule.min)
```

```
{
          errors.push(`${rule.field} must be at least
${rule.min}`);
        }
        if (rule.max !== undefined && value > rule.max)
{
          errors.push(`${rule.field} must be at most
${rule.max}`);
        }
      }
    }

    if (errors.length > 0) {
      return res.status(400).json({ errors });
    }

    next();
  };
};

// Usage
app.post('/api/users',
  validate([
    { field: 'email', type: 'email', required: true },
    { field: 'password', type: 'string', required:
true, minLength: 6 },
    { field: 'name', type: 'string', required: true,
minLength: 2, maxLength: 50 },
    { field: 'age', type: 'number', min: 18, max: 120 }
  ]),
  async (req, res) => {
    // Handle valid request
```

```
  }
);
```

60. Implement request logging middleware

Answer:

```
import { Request, Response, NextFunction } from
'express';
import fs from 'fs';
import path from 'path';

// Simple console logger
const simpleLogger = (req: Request, res: Response,
next: NextFunction) => {
  const start = Date.now();

  res.on('finish', () => {
    const duration = Date.now() - start;
    console.log(
      `[${new Date().toISOString()}] ${req.method}
${req.originalUrl} ${res.statusCode} - ${duration}ms`
    );
  });

  next();
};

// Advanced file logger
const fileLogger = (req: Request, res: Response, next:
NextFunction) => {
```

```
  const start = Date.now();
  const logDir = path.join(__dirname, 'logs');

  if (!fs.existsSync(logDir)) {
    fs.mkdirSync(logDir);
  }

  res.on('finish', () => {
    const duration = Date.now() - start;
    const logEntry = {
      timestamp: new Date().toISOString(),
      method: req.method,
      url: req.originalUrl,
      status: res.statusCode,
      duration: `${duration}ms`,
      userAgent: req.get('user-agent'),
      ip: req.ip
    };

    const logFile = path.join(logDir, `${new
Date().toISOString().split('T')[0]}.log`);
    fs.appendFileSync(logFile, JSON.stringify(logEntry)
+ '\n');
  });

  next();
};

// Using Morgan (popular logging library)
import morgan from 'morgan';

app.use(morgan('combined')); // Apache combined format
```

```
// or
app.use(morgan('dev')); // Concise colored output for
development

// Custom Morgan format
app.use(morgan(':method :url :status :response-time ms
- :res[content-length]'));
```

Additional Important Topics

61. What is `dotenv` and why use it?

Answer:

```
// .env file
PORT=3000
DATABASE_URL=postgresql://user:password@localhost:5432/
mydb
JWT_SECRET=your-secret-key
NODE_ENV=development

// Load environment variables
import dotenv from 'dotenv';
dotenv.config();

// Access variables
const PORT = process.env.PORT || 3000;
const dbUrl = process.env.DATABASE_URL;
```

```
// Never commit .env to git!
// Add to .gitignore
```

62. What is Connection Pooling?

Answer:

Connection pooling reuses database connections instead of creating new ones for each request.

```
import { Pool } from 'pg';

// Create pool
const pool = new Pool({
  host: 'localhost',
  database: 'mydb',
  user: 'postgres',
  password: 'password',
  max: 20, // Maximum number of clients in pool
  idleTimeoutMillis: 30000,
  connectionTimeoutMillis: 2000
});

// Use pool
const result = await pool.query('SELECT * FROM users');

// Pool automatically manages connections
```

63. What is the difference between `PUT` and `PATCH`?

Answer:

- **PUT**: Replace entire resource
- **PATCH**: Partial update

```
// PUT - must provide all fields
app.put('/api/users/:id', async (req, res) => {
  const { email, name, age, address } = req.body; //
All required
  await pool.query(
    'UPDATE users SET email = $1, name = $2, age = $3,
address = $4 WHERE id = $5',
    [email, name, age, address, req.params.id]
  );
});

// PATCH - provide only fields to update
app.patch('/api/users/:id', async (req, res) => {
  const updates = req.body;
  const fields = Object.keys(updates);

  const setClause = fields.map((field, i) => `${field}
= $${i + 1}`).join(', ');
  const values = [...Object.values(updates),
req.params.id];

  await pool.query(
    `UPDATE users SET ${setClause} WHERE id =
$${values.length}`,
    values
  );
});
```

64. How do you handle file uploads?

Answer:

```
import multer from 'multer';
import path from 'path';

// Configure storage
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads/');
  },
  filename: (req, file, cb) => {
    const uniqueSuffix = Date.now() + '-' +
Math.round(Math.random() * 1E9);
    cb(null, file.fieldname + '-' + uniqueSuffix +
path.extname(file.originalname));
  }
});

// File filter
const fileFilter = (req: any, file: any, cb: any) => {
  const allowedTypes = /jpeg|jpg|png|gif/;
  const extname =
allowedTypes.test(path.extname(file.originalname).toLow
erCase());
  const mimetype = allowedTypes.test(file.mimetype);

  if (extname && mimetype) {
    cb(null, true);
  } else {
    cb(new Error('Only images are allowed'));
```

```
  }
};

const upload = multer({
  storage,
  limits: { fileSize: 5 * 1024 * 1024 }, // 5MB
  fileFilter
});

// Single file
app.post('/api/upload', upload.single('avatar'), (req,
res) => {
  if (!req.file) {
    return res.status(400).json({ error: 'No file
uploaded' });
  }
  res.json({ filename: req.file.filename, path:
req.file.path });
});

// Multiple files
app.post('/api/upload-multiple', upload.array('photos',
5), (req, res) => {
  res.json({ files: req.files });
});
```

65. What is `async` vs `sync` in Node.js?

Answer:

```
import fs from 'fs';

// Synchronous - blocks execution
const dataSync = fs.readFileSync('file.txt', 'utf8');
console.log(dataSync);
console.log('After read'); // Waits for file read

// Asynchronous with callback
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
console.log('After read'); // Executes immediately

// Asynchronous with Promises
import { promises as fsPromises } from 'fs';

const dataAsync = await fsPromises.readFile('file.txt',
'utf8');
console.log(dataAsync);
```

Best Practices & Tips

General Tips:

1. Always use TypeScript for type safety
2. Use environment variables for configuration
3. Implement proper error handling

4. Use async/await instead of callbacks
5. Validate all user input
6. Hash passwords before storing
7. Use parameterized queries to prevent SQL injection
8. Implement rate limiting
9. Use HTTPS in production
10. Keep dependencies updated
11. Write tests for your code
12. Use connection pooling for databases
13. Log errors and important events
14. Handle CORS properly
15. Use proper HTTP status codes

Code Organization:

```
project/
├── src/
│   ├── config/
│   │   └── database.ts
│   ├── controllers/
│   │   └── userController.ts
│   ├── middleware/
│   │   ├── auth.ts
│   │   └── errorHandler.ts
│   ├── models/
│   │   └── User.ts
│   ├── routes/
```

```
│   │   └── userRoutes.ts
│   ├── services/
│   │   └── userService.ts
│   ├── utils/
│   │   └── validation.ts
│   └── app.ts
├── tests/
├── .env
├── .gitignore
├── package.json
└── tsconfig.json
```

Summary

This guide covers the most frequently asked questions for junior backend developers with Node.js, Express.js, MySQL/PostgreSQL, and TypeScript. Make sure to:

1. **Understand the fundamentals** - Event loop, async/await, promises
2. **Master Express.js** - Middleware, routing, error handling
3. **Know SQL basics** - CRUD operations, joins, indexes
4. **Learn TypeScript** - Types, interfaces, generics
5. **Implement security** - Authentication, authorization, input validation
6. **Write tests** - Unit tests, integration tests
7. **Follow best practices** - Code organization, error handling, logging

Remember to practice by building actual projects and solving coding challenges. Good luck with your interviews!