MERN Stack Interview Guide - Advanced Level

For Candidates with 1.5 Years Experience

MongoDB Questions

1. Explain indexing in MongoDB and when would you use compound indexes?

Answer: Indexes are data structures that improve query performance by allowing MongoDB to locate documents without scanning entire collections. Compound indexes are indexes on multiple fields, useful when queries filter or sort by multiple fields. For example, `db.users.createIndex({email: 1, status: 1})` optimizes queries filtering by both email and status. Use compound indexes when your queries consistently use the same field combinations.

2. What is the aggregation pipeline? Explain $lookup, $unwind, and $group.

Answer: The aggregation pipeline processes documents through multiple stages. `$lookup` performs left outer joins between collections. `$unwind` deconstructs array fields into separate documents. `$group` groups documents by a specified key and performs accumulations. Example:

```
db.orders.aggregate([
  { $lookup: { from: "products", localField: "productId", foreignField: "_id", as: "product" } },
  { $unwind: "$product" },
  { $group: { _id: "$userId", totalSpent: { $sum: "$product.price" } } }
])
```

3. What's the difference between embedding and referencing in MongoDB schema design?

Answer: Embedding stores related data within a single document, providing atomic operations and better read performance. Referencing stores related data in separate collections with document IDs linking them, reducing duplication and improving write performance. Use embedding for one-to-few relationships with frequently accessed data together. Use referencing for one-to-many or many-to-many relationships where data is accessed independently.

4. How do you handle transactions in MongoDB?

Answer: MongoDB supports ACID transactions using sessions. For replica sets (MongoDB 4.0+) and sharded clusters (4.2+), you can use multi-document transactions:

```
const session = await mongoose.startSession();
session.startTransaction();
try {
  await User.updateOne({_id: userId}, {$inc: {balance: -100}}, {session});
  await Transaction.create([{userId, amount: -100}], {session});
  await session.commitTransaction();
} catch (error) {
  await session.abortTransaction();
} finally {
  session.endSession();
}
```

5. What are the read and write concerns in MongoDB?

Answer: Read concern determines the consistency level for read operations (local, available, majority, linearizable). Write concern specifies acknowledgment level for writes (w: 0, 1, majority). For critical operations, use `{w: "majority", j: true}` to ensure data is written to the journal on a majority of replica set members before acknowledging.

Express.js Questions

6. Explain middleware execution order and error-handling middleware in Express.

Answer: Middleware executes in the order it's defined. Error-handling middleware must have four parameters `(err, req, res, next)` and should be defined after all other middleware and routes. Regular middleware has three parameters `(req, res, next)`. Errors in async functions must be caught and passed to `next(error)` or use express-async-errors package.

7. How do you implement rate limiting in Express?

Answer: Use packages like express-rate-limit:

```
const rateLimit = require('express-rate-limit');
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // limit each IP to 100 requests per windowMs
  message: 'Too many requests from this IP'
});
app.use('/api/', limiter);
```

For distributed systems, use Redis with rate-limit-redis store to share rate limit data across instances.

8. What's the difference between app.use() and app.all()?

Answer: `app.use()` matches any HTTP method and path prefix (e.g., '/api' matches '/api/users'). `app.all()` matches any HTTP method but requires exact path match unless using wildcards. `app.use()` is typically for middleware, while `app.all()` is for route handlers applying to all methods.

9. How do you handle file uploads in Express?

Answer: Use multer middleware for multipart/form-data:

```
const multer = require('multer');
const storage = multer.diskStorage({
  destination: './uploads/',
  filename: (req, file, cb) => {
    cb(null, Date.now() + '-' + file.originalname);
  }
});
const upload = multer({
  storage,
  limits: { fileSize: 5 * 1024 * 1024 },
  fileFilter: (req, file, cb) => {
    if (file.mimetype.startsWith('image/')) cb(null, true);
    else cb(new Error('Only images allowed'));
  }
});
app.post('/upload', upload.single('file'), (req, res) => {});
```

10. Explain CORS and how to configure it properly in Express.

Answer: CORS (Cross-Origin Resource Sharing) allows controlled access to resources from different origins. Configure with cors package:

```
const cors = require('cors');
app.use(cors({
  origin: ['http://localhost:3000', 'https://myapp.com'],
  credentials: true,
  methods: ['GET', 'POST', 'PUT', 'DELETE'],
  allowedHeaders: ['Content-Type', 'Authorization']
}));
```

For preflight requests, Express automatically handles OPTIONS requests when using cors middleware.

React Questions

11. Explain React hooks lifecycle and when useEffect runs.

Answer: useEffect runs after render completes. With empty dependency array `[]`, it runs once after initial render (like componentDidMount). With dependencies `[dep1, dep2]`, it runs when those values change. Without dependency array, it runs after every render. The cleanup function runs before the effect re-runs or component unmounts.

12. What is the difference between useMemo and useCallback?

Answer: `useMemo` memoizes computed values to avoid expensive recalculations: `useMemo(() => computeExpensive(a, b), [a, b])`. `useCallback` memoizes function references to prevent child re-renders: `useCallback(() => doSomething(a, b), [a, b])`. Use useMemo for expensive calculations, useCallback to stabilize function references passed to optimized child components.

13. How do you optimize React application performance?

Answer:

- Use React.memo for component memoization
- Implement code splitting with React.lazy and Suspense
- Use useCallback/useMemo to prevent unnecessary re-renders
- Virtualize long lists with react-window or react-virtualized
- Avoid inline object/array creation in render
- Use production build for deployment
- Implement proper key props in lists
- Lazy load images and components

14. Explain Context API and when to use it vs Redux.

Answer: Context API provides component tree-wide state without prop drilling. Use Context for theme, auth, or locale data that changes infrequently. Use Redux for complex state logic, time-travel debugging, middleware requirements, or when you need strict unidirectional data flow. Context can cause unnecessary re-renders if not structured properly, whereas Redux with proper selectors optimizes rendering better.

15. What are controlled vs uncontrolled components?

Answer: Controlled components have React state as the "single source of truth" - form inputs receive values from state and update via onChange handlers. Uncontrolled components store their own state internally in the DOM, accessed via refs. Use controlled for validation, conditional rendering, or dynamic inputs. Use uncontrolled for simple forms or file inputs (which must be uncontrolled).

16. Explain React reconciliation and the Virtual DOM.

Answer: Reconciliation is React's diffing algorithm that updates the DOM efficiently. React maintains a Virtual DOM (lightweight copy of actual DOM). When state changes, React creates a new Virtual DOM tree, compares it with the previous one (diffing), and calculates minimal DOM updates needed. Keys help React identify which items changed. This process is why React is fast - batch updates and minimal DOM manipulation.

17. How do you handle forms in React? Explain Formik or React Hook Form.

Answer: React Hook Form is performant (uncontrolled components with refs) and has minimal re-renders:

```
import { useForm } from 'react-hook-form';
const { register, handleSubmit, formState: { errors } } = useForm();
const onSubmit = data => console.log(data);
<form onSubmit={handleSubmit(onSubmit)}>
  <input {...register('email', { required: true, pattern: /^\S+@\S+$/i })} />
  {errors.email && <span>Invalid email</span>}
</form>
```

It integrates with validation libraries like Yup/Zod for schema validation.

18. What are React Portals and when would you use them?

Answer: Portals render children into a DOM node outside the parent component hierarchy: `ReactDOM.createPortal(child, container)`. Use cases include modals, tooltips, dropdowns that need to break out of parent's overflow/z-index constraints while maintaining React event bubbling through the component tree.

19. Explain custom hooks and provide an example.

Answer: Custom hooks extract reusable stateful logic. Example:

```
function useLocalStorage(key, initialValue) {
  const [value, setValue] = useState(() => {
    const item = localStorage.getItem(key);
    return item ? JSON.parse(item) : initialValue;
  });

  useEffect(() => {
    localStorage.setItem(key, JSON.stringify(value));
  }, [key, value]);

  return [value, setValue];
}
```

Usage: `const [theme, setTheme] = useLocalStorage('theme', 'light');`

20. How do you implement code splitting in React?

Answer: Use React.lazy for dynamic imports and Suspense for fallback UI:

```
const Dashboard = React.lazy(() => import('./Dashboard'));

function App() {
  return (
    <Suspense fallback={<Spinner />}>
      <Dashboard />
    </Suspense>
  );
}
```

Also use route-based code splitting with React Router. Webpack automatically creates separate bundles for lazy-loaded components.

Node.js Questions

21. Explain the Node.js event loop and its phases.

Answer: The event loop has six phases: timers (setTimeout/setInterval callbacks), pending callbacks (I/O callbacks deferred to next iteration), idle/prepare (internal use), poll (retrieve new I/O events), check (setImmediate callbacks), close callbacks. Process.nextTick and Promise microtasks run between phases. Understanding this helps debug timing issues and optimize async code.

22. What are streams in Node.js? Explain types and use cases.

Answer: Streams are collections of data that might not be available all at once. Four types: Readable (read data), Writable (write data), Duplex (both), Transform (modify data while reading/writing). Use streams for large files to avoid memory issues:

```
const fs = require('fs');
const readStream = fs.createReadStream('large.txt');
const writeStream = fs.createWriteStream('copy.txt');
readStream.pipe(writeStream);
```

Streams emit events: data, end, error. They're memory-efficient for processing large datasets.

23. How do you handle environment variables in Node.js?

Answer: Use dotenv package to load variables from .env file:

```
require('dotenv').config();
const dbUrl = process.env.DATABASE_URL;
```

Never commit .env to version control. Use different .env files for environments (.env.development, .env.production). For production, use environment variables set by hosting platform or container orchestration.

24. Explain clustering in Node.js.

Answer: Node.js runs single-threaded by default. Clustering spawns multiple Node processes to utilize multi-core systems:

```
const cluster = require('cluster');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
} else {
  require('./server');
}
```

Master process distributes incoming connections among workers. Use PM2 in production for automatic clustering and process management.

25. What's the difference between process.nextTick() and setImmediate()?

Answer: `process.nextTick()` executes callback after current operation completes, before event loop continues (microtask queue). `setImmediate()` executes in the check phase of the event loop. nextTick has higher priority and can starve the event loop if used recursively. Use setImmediate for deferring execution to avoid blocking.

26. How do you implement caching in Node.js?

Answer: Use Redis for distributed caching:

```
const redis = require('redis');
const client = redis.createClient();

async function getUser(id) {
  const cached = await client.get(`user:${id}`);
  if (cached) return JSON.parse(cached);

  const user = await User.findById(id);
  await client.setEx(`user:${id}`, 3600, JSON.stringify(user));
  return user;
}
```

Also use node-cache for in-memory caching, but it doesn't share across instances. Implement cache invalidation strategies (TTL, LRU).

27. Explain worker threads in Node.js.

Answer: Worker threads allow parallel JavaScript execution for CPU-intensive tasks:

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

const worker = new Worker('./worker.js', { workerData: { num: 5 } });
worker.on('message', result => console.log(result));
worker.on('error', error => console.error(error));
```

Use for image processing, encryption, or complex calculations. Different from clustering - workers share memory, clusters don't.

28. How do you implement logging in Node.js applications?

Answer: Use Winston or Pino for structured logging:

```
const winston = require('winston');
const logger = winston.createLogger({
  level: 'info',
  format: winston.format.json(),
  transports: [
    new winston.transports.File({ filename: 'error.log', level: 'error' }),
    new winston.transports.File({ filename: 'combined.log' })
  ]
});

if (process.env.NODE_ENV !== 'production') {
  logger.add(new winston.transports.Console());
}
```

Log levels: error, warn, info, debug. In production, send logs to centralized systems like ELK stack or CloudWatch.

29. What are memory leaks in Node.js and how do you detect them?

Answer: Common causes: global variables, forgotten timers/listeners, closures holding references, large caches. Detect using:

- Chrome DevTools memory profiler
- `process.memoryUsage()` monitoring
- Heap snapshots with `--inspect` flag
- Tools like clinic.js or node-memwatch

Prevention: remove event listeners, clear timers, implement proper cleanup, use WeakMap for caches, limit array/object growth.

30. Explain the difference between spawn, exec, and fork in child_process.

Answer:

- `spawn`: Streams data from child process, suitable for large data. Returns EventEmitter.
- `exec`: Buffers output, returns callback with stdout/stderr. Suitable for small output.
- `fork`: Special case of spawn for Node processes, enables IPC between parent and child.

```
const { spawn, exec, fork } = require('child_process');
const ls = spawn('ls', ['-lh']);
exec('ls -lh', (err, stdout) => {});
const child = fork('./script.js');
```

Security & Authentication

31. How do you implement JWT authentication in MERN stack?

Answer:

```
// Generate token
const jwt = require('jsonwebtoken');
const token = jwt.sign({ userId: user._id }, process.env.JWT_SECRET, { expiresIn: '7d' });
```

```
// Verify middleware
const auth = async (req, res, next) => {
  try {
    const token = req.header('Authorization').replace('Bearer ', '');
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    const user = await User.findById(decoded.userId);
    if (!user) throw new Error();
    req.user = user;
    next();
  } catch (error) {
    res.status(401).json({ error: 'Please authenticate' });
  }
};
```

Store tokens in httpOnly cookies or localStorage (less secure). Implement refresh tokens for long-lived sessions.

32. What are common security vulnerabilities and how do you prevent them?

Answer:

- **XSS**: Sanitize user input, use React's built-in escaping, Content Security Policy headers
- **CSRF**: Use CSRF tokens, SameSite cookies, verify origin headers
- **SQL/NoSQL Injection**: Use parameterized queries, validate input, use Mongoose schemas
- **Sensitive Data Exposure**: Encrypt data at rest, use HTTPS, never log sensitive info
- **Broken Authentication**: Strong password policies, rate limiting, MFA, secure session management
- Use Helmet.js for security headers, rate-limit API endpoints, validate all inputs

33. How do you hash passwords securely?

Answer: Use bcrypt with salt rounds (10-12):

```
const bcrypt = require('bcrypt');

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

// Verification
const isMatch = await bcrypt.compare(password, user.hashedPassword);
```

Never store plain text passwords. Bcrypt is intentionally slow to prevent brute force attacks. Salt is automatically handled by bcrypt.

34. Explain OAuth 2.0 flow and how to implement social login.

Answer: OAuth 2.0 flow: User clicks "Login with Google" → Redirect to provider → User authorizes → Provider redirects back with authorization code → Exchange code for access token → Use token to fetch user info. Implement using Passport.js:

```
const passport = require('passport');
const GoogleStrategy = require('passport-google-oauth20').Strategy;

passport.use(new GoogleStrategy({
  clientID: process.env.GOOGLE_CLIENT_ID,
  clientSecret: process.env.GOOGLE_CLIENT_SECRET,
  callbackURL: '/auth/google/callback'
}, async (accessToken, refreshToken, profile, done) => {
  // Find or create user
}));
```

35. What is HTTPS and why is it important?

Answer: HTTPS encrypts data between client and server using TLS/SSL, preventing man-in-the-middle attacks. It ensures data integrity, authentication, and confidentiality. Implement using Let's Encrypt certificates. In production, use reverse proxies (Nginx) to handle SSL termination. Always redirect HTTP to HTTPS and use HSTS headers.

Advanced Topics

36. Explain microservices architecture vs monolithic architecture.

Answer: Monolithic: Single codebase, easier to develop initially, simpler deployment, but scaling requires scaling entire app. Microservices: Independent services communicating via APIs, scale independently, technology agnostic, but complex deployment and network overhead. Use microservices for large teams and independent scaling needs. Start monolithic, break into microservices when needed.

37. How do you implement real-time communication in MERN stack?

Answer: Use Socket.io for WebSocket connections:

```
// Server
const io = require('socket.io')(server, { cors: { origin: '*' } });
io.on('connection', (socket) => {
  socket.on('message', (data) => {
    io.emit('message', data); // Broadcast to all
  });
});

// Client
import io from 'socket.io-client';
const socket = io('http://localhost:5000');
socket.on('message', (data) => console.log(data));
socket.emit('message', { text: 'Hello' });
```

For scalability, use Redis adapter to sync across multiple server instances.

38. What is GraphQL and how does it differ from REST?

Answer: GraphQL is a query language for APIs allowing clients to request exactly what they need. REST uses multiple endpoints with fixed responses. GraphQL advantages: single endpoint, no over/under-fetching, strongly typed schema, better for mobile apps. REST is simpler for CRUD operations. Implement with Apollo Server and Apollo Client in React.

39. How do you implement pagination in MERN stack?

Answer:

```
// Backend
const page = parseInt(req.query.page) || 1;
const limit = parseInt(req.query.limit) || 10;
const skip = (page - 1) * limit;

const users = await User.find()
  .skip(skip)
  .limit(limit)
  .sort({ createdAt: -1 });
const total = await User.countDocuments();

res.json({ users, total, page, pages: Math.ceil(total / limit) });

// Frontend with React
const [page, setPage] = useState(1);
const fetchUsers = async () => {
  const { data } = await axios.get(`/api/users?page=${page}&limit=10`);
  setUsers(data.users);
};
```

For large datasets, implement cursor-based pagination for better performance.

40. What is Docker and how do you containerize a MERN application?

Answer: Docker packages applications with dependencies into containers. Create Dockerfile:

```
FROM node:16-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 5000
CMD ["node", "server.js"]
```

Docker-compose for full stack:

```
version: '3'
services:
  mongo:
    image: mongo
  backend:
    build: ./backend
    ports: ["5000:5000"]
  frontend:
    build: ./frontend
    ports: ["3000:3000"]
```

Benefits: consistent environments, easy deployment, isolation, scalability.

41. How do you implement search functionality in MongoDB?

Answer: Use MongoDB text indexes for full-text search:

```
// Create text index
db.products.createIndex({ name: "text", description: "text" });

// Search
const products = await Product.find(
  { $text: { $search: query } },
  { score: { $meta: "textScore" } }
).sort({ score: { $meta: "textScore" } });
```

For advanced search, use Elasticsearch or Algolia. Implement fuzzy search, filters, and autocomplete for better UX.

42. What are webhooks and how do you implement them?

Answer: Webhooks are HTTP callbacks triggered by events, allowing real-time communication between services. Implementation:

```
// Webhook endpoint
app.post('/webhook', async (req, res) => {
  const signature = req.headers['x-signature'];
  // Verify signature to ensure request is from trusted source
  if (!verifySignature(req.body, signature)) {
    return res.status(401).send('Unauthorized');
  }

  // Process event
  await processEvent(req.body);
  res.status(200).send('OK');
});
```

Use webhooks for payment confirmations (Stripe), notifications, or third-party integrations. Implement retry logic and idempotency.

43. How do you handle file storage in production?

Answer: Use cloud storage services:

- AWS S3: Scalable, reliable, CDN integration with CloudFront
- Google Cloud Storage or Azure Blob Storage

```
const AWS = require('aws-sdk');
const s3 = new AWS.S3();

const uploadToS3 = async (file) => {
  const params = {
    Bucket: process.env.S3_BUCKET,
    Key: `uploads/${Date.now()}-${file.originalname}`,
    Body: file.buffer,
    ContentType: file.mimetype,
    ACL: 'public-read'
  };
  const result = await s3.upload(params).promise();
  return result.Location;
};
```

Never store files on server filesystem in production - it's not scalable across multiple instances.

44. What is server-side rendering (SSR) and when would you use it?

Answer: SSR renders React components on the server and sends HTML to client. Benefits: better SEO, faster initial page load, improved performance on slow devices. Use Next.js for SSR in React:

```
export async function getServerSideProps(context) {
  const data = await fetchData();
  return { props: { data } };
}
```

Use SSR for content-heavy sites, public pages requiring SEO, or when initial load performance is critical. Use CSR (client-side rendering) for dashboards and authenticated apps.

45. How do you implement testing in MERN stack?

Answer:

- **Backend**: Use Jest and Supertest for API testing

```
const request = require('supertest');
test('GET /api/users', async () => {
  const res = await request(app).get('/api/users');
  expect(res.status).toBe(200);
  expect(res.body).toHaveProperty('users');
});
```

- **Frontend**: Use Jest and React Testing Library

```
import { render, screen } from '@testing-library/react';
test('renders button', () => {
  render(<Button>Click</Button>);
  expect(screen.getByText('Click')).toBeInTheDocument();
});
```

- **E2E**: Use Cypress or Playwright Implement unit tests, integration tests, and E2E tests for critical user flows.

46. What is CI/CD and how do you implement it?

Answer: CI/CD automates testing and deployment. Use GitHub Actions, GitLab CI, or Jenkins:

```
name: CI/CD
on: [push]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - run: npm install
      - run: npm test
  deploy:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - run: npm run build
      - run: deploy to production
```

Automate: linting, testing, building, deployment. Use feature branches, require PR reviews, run tests before merge.

47. How do you monitor and debug production applications?

Answer: Use monitoring tools:

- **Application monitoring**: New Relic, Datadog, or PM2
- **Error tracking**: Sentry for frontend and backend errors
- **Logging**: ELK stack (Elasticsearch, Logstash, Kibana) or CloudWatch
- **Performance**: Lighthouse for frontend, APM tools for backend

```
const Sentry = require('@sentry/node');
Sentry.init({ dsn: process.env.SENTRY_DSN });

app.use(Sentry.Handlers.requestHandler());
// routes
app.use(Sentry.Handlers.errorHandler());
```

Set up alerts for error rates, response times, server health. Use distributed tracing for microservices.

48. What are design patterns commonly used in MERN stack?

Answer:

- **MVC**: Separates concerns (Models, Views, Controllers)
- **Repository Pattern**: Abstracts data access layer
- **Factory Pattern**: Creates objects without specifying exact class
- **Singleton**: Single instance (database connection)
- **Observer**: Event-driven architecture (EventEmitter)
- **Middleware Pattern**: Express middleware chain
- **Higher-Order Components/Hooks**: React composition Use patterns to write maintainable, testable, scalable code.

49. How do you optimize MongoDB queries?

Answer:

- Create indexes on frequently queried fields
- Use projection to return only needed fields: `User.find().select('name email')`
- Use lean() for read-only queries: `User.find().lean()`
- Limit results and use pagination
- Use aggregation pipeline instead of multiple queries
- Avoid $where and JavaScript expressions
- Monitor slow queries with MongoDB profiler: `db.setProfilingLevel(1, { slowms: 100 })`
- Use explain() to analyze query performance: `db.collection.find().explain()`

50. What are the best practices for structuring a MERN application?

Answer:

```
backend/
 ├── config/        # Configuration files
 ├── controllers/   # Route handlers
 ├── models/        # Database models
 ├── routes/        # API routes
 ├── middleware/    # Custom middleware
 ├── utils/         # Helper functions
 ├── services/      # Business logic
 └── server.js

frontend/
 ├── public/
 ├── src/
 │   ├── components/  # Reusable components
 │   ├── pages/       # Page components
 │   ├── hooks/       # Custom hooks
 │   ├── context/     # Context providers
 │   ├── services/    # API calls
 │   ├── utils/       # Helper functions
 │   └── App.js
```

Follow: separation of concerns, DRY principle, consistent naming, error handling, environment variables, documentation, code reviews, testing.

Tips for Interview Success

1. **Explain your thought process** - Don't just answer, explain why
2. **Ask clarifying questions** - Shows attention to detail
3. **Discuss trade-offs** - Every solution has pros and cons
4. **Use real project examples** - Reference your actual work experience
5. **Be honest about gaps** - Say "I haven't used that but I know..." or "I'd research..."
6. **Think aloud** - Interviewers want to understand your problem-solving approach
7. **Stay updated** - Know latest versions and features
8. **Practice coding** - Use LeetCode, HackerRank for algorithm questions

Good luck with your interview! 🚀