ChatGPT

Node.js, React, Kafka & TypeScript - Interview Q&A (1–2 min speech format)

---

1. Callback Hell in Node.js and Solutions

Callback hell refers to deeply nested callbacks in Node.js due to asynchronous operations, making code unreadable and difficult to maintain.

Example of callback hell:

```
fs.readFile('file1.txt', (err, data1) => {
  fs.readFile('file2.txt', (err, data2) => {
    fs.readFile('file3.txt', (err, data3) => {
      // Do something
    });
  });
});
```

Solution: Promises and async/await:

```
const readFile = util.promisify(fs.readFile);
async function readFiles() {
  const data1 = await readFile('file1.txt');
  const data2 = await readFile('file2.txt');
  const data3 = await readFile('file3.txt');
}
```

Benefits: - Flatter structure - Better error handling - Easier to read and maintain

---

2. Error Handling in Node.js

Synchronous errors are caught using try-catch. **Asynchronous errors** (e.g., in callbacks or Promises) must be handled using error callbacks or `.catch()`.

Global handlers:

```
process.on('uncaughtException', (err) => { ... });
process.on('unhandledRejection', (reason, promise) => { ... });
```

Best practices: - Always handle errors in async operations - Use centralized error-handling middleware in Express - Avoid suppressing errors silently

---

3. Node.js Concurrency & libuv

Node.js is **single-threaded**, but it handles concurrency using the **libuv** library. libuv offloads I/O operations (file system, network, DNS, etc.) to the **thread pool** or OS kernel.

Be cautious with **CPU-bound tasks** like encryption or large JSON parsing — they can **block the event loop**.

Solution: Offload heavy tasks to background workers using libraries like `worker_threads` or message queues.

---

4. React Re-renders: React.memo vs PureComponent

- `React.memo` is for **functional components** and prevents re-renders unless props change.
- `PureComponent` is for **class components**, and does shallow prop and state comparison.

Use these when: - You have large trees and performance issues - Props are stable and don't change frequently

Caution: Avoid premature optimization—only use when performance bottlenecks are proven.

---

5. React Hooks Overview

Hooks let you use state and lifecycle in **functional components**. - `useState`: manage local state - `useEffect`: handle side effects (e.g., fetching data, subscriptions)

Example:

```
const [count, setCount] = useState(0);
useEffect(() => { document.title = `Count: ${count}`; }, [count]);
```

Hooks replaced many lifecycle methods like `componentDidMount` and `componentDidUpdate`.

---

6. Kafka Fault Tolerance

Kafka ensures **high availability** via: - **Replication**: each partition is replicated across brokers - **Leader-follower**: writes go to leader, followers sync - **Zookeeper**: manages broker metadata (now moving to KRaft)

If a broker crashes, replicas take over with no data loss.

---

7. Security in Node.js Applications

Common vulnerabilities: - XSS - SQL/NoSQL Injection - CSRF

Best practices: - Use `helmet.js` for HTTP headers - Validate and sanitize input (e.g., using `express-validator`) - Use HTTPS - Avoid exposing sensitive data

Environment variables must be managed securely (e.g., `.env`, `dotenv`).

---

8. CommonJS vs ES Modules

CommonJS:

```
const fs = require('fs');
module.exports = myFunc;
```

ES Modules:

```
import fs from 'fs';
export default myFunc;
```

Differences: - ESM is static and supports tree-shaking - CJS is dynamic and widely used in legacy code - Node.js now supports both, but mixing requires care

---

9. Azure SQL Backup & Disaster Recovery

Steps: - Enable **automated backups** (point-in-time restore) - Use **Geo-replication** for DR - Periodically export to Blob Storage - Test **restoration plan** regularly

Tools: Azure Portal, Azure CLI, or ARM templates.

---

10. React Context API vs Prop Drilling

Prop drilling means passing props through many levels. **Context API** provides a global way to share state:

```
const MyContext = createContext();
<MyContext.Provider value={value}>...</MyContext.Provider>
```

When to use Context: - Auth state - Theme settings - Locale

For complex state, consider Redux or Zustand.

---

11. Set up Azure VM & Remote Access

Steps: 1. Create VM in Azure Portal 2. Choose OS (Windows/Linux) 3. Open necessary ports (e.g., RDP 3389 or SSH 22) 4. Download RDP file or use SSH key 5. Connect using Remote Desktop or Terminal

Use NSG (Network Security Groups) for access control.

---

12. Consume Kafka Messages

Example with `kafkajs`:

```
const { Kafka } = require('kafkajs');
const kafka = new Kafka({ clientId: 'my-app', brokers: ['localhost:9092'] });
const consumer = kafka.consumer({ groupId: 'test-group' });

await consumer.connect();
await consumer.subscribe({ topic: 'my-topic', fromBeginning: true });
consumer.run({ eachMessage: async ({ message }) =>
console.log(message.value.toString()) });
```

---

13. MongoDB Document-Oriented Database

MongoDB stores data as **documents** in **collections** using JSON-like BSON.

Benefits: - Schema flexibility - Better for nested/complex data - Ideal for real-time apps, logs, etc.

Example:

```
{
  "name": "John",
  "orders": [{ "item": "book", "qty": 2 }]
}
```

---

14. TypeScript `keyof` Operator

`keyof` lets you get the union of an object's keys.

```
type User = { name: string; age: number };
type Keys = keyof User; // 'name' | 'age'

function getProp<T, K extends keyof T>(obj: T, key: K): T[K] {
```

```
  return obj[key];
}
```

Useful for building **type-safe utilities**.

---

15. Environment-Specific Config in Node.js

Use **environment variables**:

```
NODE_ENV=production DB_HOST=prod.example.com
```

Use libraries like `dotenv`, `config`, or `rc`.

Structure: - `.env.development` - `.env.production`

Best practices: - Don't hardcode secrets - Use secret managers (e.g., Azure Key Vault, AWS Secrets Manager)
