ChatGPT

Node.js, React, Kafka & TypeScript - Interview Q&A (Speech Format with Code)

---

1. Callback Hell in Node.js and Solutions

In Node.js, we often handle asynchronous operations like reading files or querying databases. When callbacks are nested too deeply, it results in what's called **callback hell**, making the code hard to read and maintain.

For example:

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

This pyramid of doom can be refactored using Promises and async/await for cleaner code:

```
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');
  // process data
}
```

This version reads like synchronous code, improving maintainability and reducing nesting.

---

2. Error Handling in Node.js

In Node.js, handling errors correctly is crucial. Synchronous errors can be caught with try-catch blocks, but asynchronous ones require handling in callbacks or Promises.

Example of a try-catch:

```
try {
  let data = fs.readFileSync('file.txt');
} catch (err) {
  console.error('Error:', err);
}
```

With Promises:

```
readFile('file.txt')
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

Node also provides global events:

```
process.on('uncaughtException', err => console.error('Uncaught:',
err));
process.on('unhandledRejection', reason =>
console.error('Unhandled:', reason));
```

These should be used for logging and alerts—not to recover the app.

---

3. Node.js Concurrency and libuv

Although Node.js uses a single-threaded model, it achieves concurrency using the **libuv** library. libuv offloads operations like file I/O or DNS lookup to background threads.

For example, `fs.readFile` doesn't block the event loop. But if we use synchronous methods or run CPU-intensive tasks, we can block the event loop.

To avoid this, we can use `worker_threads` for CPU-bound work, or delegate tasks to external systems when needed.

---

4. React.memo vs PureComponent

In React, unnecessary re-renders can impact performance. `React.memo` is used with functional components to avoid re-renders if props haven't changed. Similarly, `PureComponent` works with class components.

Example with `React.memo`:

```
const MyComponent = React.memo(({ name }) => <div>{name}</div>);
```

It does a shallow comparison of props.

Use these optimizations when your component renders large data or receives stable props over time.

---

5. React Hooks

React hooks simplify writing functional components.

- `useState` handles local state
- `useEffect` replaces lifecycle methods like `componentDidMount`

Example:

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

This approach makes components more modular and easier to test.

---

6. Kafka Fault Tolerance

Kafka ensures high availability using partition replication. Each partition has a leader and multiple followers. If a broker fails, a follower takes over.

Zookeeper manages this metadata—though newer versions are moving toward KRaft. These mechanisms make Kafka resilient to node failures and network issues.

---

7. Security in Node.js

Security in Node.js involves preventing vulnerabilities like XSS, CSRF, and injection attacks.

Best practices include: - Using `helmet.js` to set secure HTTP headers - Validating inputs with libraries like `express-validator` - Storing secrets using environment variables or vaults

Example of using helmet:

```
const helmet = require('helmet');
app.use(helmet());
```

---

8. CommonJS vs ES Modules

CommonJS uses `require`, while ES Modules use `import`. ES Modules allow static analysis, better tree-shaking, and are now fully supported in modern Node.js.

CommonJS:

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

ES Module:

```
import fs from 'fs';
```

Choose based on compatibility, but modern projects should prefer ES Modules.

---

9. Azure SQL Backup & DR

For Azure SQL, enable automatic backups, configure geo-replication for disaster recovery, and test restoring backups periodically.

You can also use the Azure CLI to export data to Blob Storage and automate alerts for failures.

---

10. React Context API vs Prop Drilling

Prop drilling happens when we pass props through many nested components. Context API solves this by providing a global value.

Example:

```
const ThemeContext = React.createContext();
<ThemeContext.Provider value={theme}>
  <App />
</ThemeContext.Provider>
```

It's ideal for auth, theming, or language settings. For more complex needs, use Redux or Zustand.

---

11. Setting Up an Azure VM

To set up a VM in Azure: - Choose an image like Ubuntu or Windows - Set authentication using SSH key or password - Configure the NSG to open ports like 22 for SSH or 3389 for RDP - Use RDP or terminal to connect

12. Kafka Consumer Example

Consuming messages from Kafka involves setting up a consumer group and subscribing to a topic.

Example using kafkajs:

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

Consumers are scalable and reliable, making them ideal for microservices.

13. MongoDB Document-Oriented Database

MongoDB stores data in flexible, JSON-like documents inside collections.

Example document:

```
{
  "name": "Alice",
  "orders": [
    { "item": "pen", "qty": 2 }
  ]
}
```

This structure is great for nested data and evolves over time without migrations.

14. TypeScript `keyof` Operator

`keyof` creates a union of object keys. It ensures type-safe property access.

Example:

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

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

It’s powerful for generic utilities and refactoring code safely.

---

15. Managing Environments in Node.js

Use environment variables for config like DB credentials, ports, or API keys. Tools like `dotenv` let you load variables from files.

Example `.env` file:

```
DB_URL=mongodb://localhost
NODE_ENV=development
```

Load with:

```
require('dotenv').config();
```

Always keep sensitive config out of code and version control.
