Capgemini- Frontend Developer

Interview Process

Round 1: HR Round
Round 2: Assignment
Round 3: Coding Round
Round 4: HR Round

Interview Questions

1. What is the life cycle method?
2. What hooks have you used?
3. What is map filter and reduce?
4. How to make JavaScript asynchronous?
5. What is the difference between redux thunk and redux saga?
6. What is the benefit of arrow function?
7. What are the hoisting new features of ES6?
8. Write code in react to call an api and display some data Promises
9. When you call three apis, how can you achieve that?
10. What are different promise apis available Custom hooks?
11. What are pure components?
12. What is react and it's advantage?
13. Call api and print data in using react
14. Agile process followed in project
15. Simple app using redux
16. Reverse given array
17. React lifecycle
18. Can you explain the difference between id and class attributes in HTML/CSS?
19. How do you declare variables in JavaScript?
20. Can you explain what a function is and how you define one in JavaScript?

SOLUTIONS

Q1: What is the life cycle method?
A1: In React, lifecycle methods are special methods that get invoked at different stages of a component's life, such as when it is being created, updated, or destroyed. The main phases are Mounting, Updating, and Unmounting. During mounting, methods like `componentDidMount` are called, allowing you to run code after the component is inserted into the DOM. During updating, methods like `componentDidUpdate` let you react to changes in props or state.

When unmounting, `componentWillUnmount` is used to clean up any resources, like timers or event listeners, before the component is removed from the DOM.

Q2: What hooks have you used?
A2: Common React hooks include `useState` for managing state within a function component, `useEffect` for handling side effects like data fetching or subscriptions, `useContext` for accessing context values without passing props down manually, `useReducer` for managing complex state logic similar to Redux, and `useRef` for persisting values across renders without triggering re-renders. Custom hooks, which are reusable pieces of logic encapsulated in a function, can also be created to abstract complex operations.

Q3: What is map, filter, and reduce?
A3: `map`, `filter`, and `reduce` are array methods in JavaScript used for transforming and processing data:

- `map`: Transforms each element in an array and returns a new array with the transformed elements. For example, doubling each number in an array.
- `filter`: Returns a new array containing only the elements that meet a specified condition. For example, filtering out all even numbers.
- `reduce`: Accumulates array elements into a single value based on a reducer function. For example, summing all numbers in an array. These methods are often used in combination for powerful data manipulation.

Q4: How to make JavaScript asynchronous?
A4: JavaScript can be made asynchronous using callbacks, Promises, and `async/await`. Callbacks were the original method, but they can lead to callback hell. Promises simplify handling asynchronous operations by chaining `.then()` and `.catch()` methods. `async/await`, introduced in ES8, provides a more readable and synchronous-looking way to write asynchronous code by using the `async` keyword to define functions and `await` to pause execution until a Promise is resolved.

Q5: What is the difference between redux thunk and redux saga?
A5: Redux Thunk and Redux Saga are middleware for managing side effects in Redux:

- Redux Thunk: Allows you to write action creators that return a function instead of an action. This function can dispatch actions and run asynchronous code. It's simple and great for basic async logic.
- Redux Saga: Uses generator functions to manage more complex side effects, such as handling complex sequences of actions, long-running processes, or better control over API calls. Redux Saga offers more power and flexibility, especially in complex applications.

Q6: What is the benefit of arrow function?
A6: Arrow functions in JavaScript provide a concise syntax and fix the context of `this` within the function. Unlike regular functions, arrow functions don't have their own `this` context; instead, they inherit `this` from the surrounding lexical context. This makes arrow functions particularly useful in scenarios where you want to preserve the context, such as in callbacks or event handlers.

Q7: What are the hoisting and new features of ES6?
A7: Hoisting in JavaScript refers to the behavior where variable and function declarations are moved to the top of their containing scope during compilation. In ES6, `let` and `const` declarations are hoisted but not initialized, which means they are not accessible until the code execution reaches the declaration (Temporal Dead Zone). ES6 introduced several new features, including arrow functions, template literals, destructuring, default parameters, `let` and `const` for block-scoped variables, and modules for better code organization.

Q8: Write code in React to call an API and display some data using Promises.
A8:

```
import React, { useState, useEffect } from 'react';

function App() {
    const [data, setData] = useState([]);

    useEffect(() => {
        fetch('https://jsonplaceholder.typicode.com/posts')
            .then(response => response.json())
            .then(data => setData(data))
            .catch(error => console.error('Error:', error));
    }, []);
```

```
    return (
        <div>
            <h1>Posts</h1>
            <ul>
                {data.map(post => (
                    <li key={post.id}>{post.title}</li>
                ))}
            </ul>
        </div>
    );
}

export default App;
```

Q9: When you call three APIs, how can you achieve that?
A9: To call three APIs in parallel, you can use `Promise.all()` which takes an array of Promises and returns a single Promise that resolves when all the input Promises have resolved. Here's an example:

```
Promise.all([
    fetch('https://api.example.com/endpoint1'),
    fetch('https://api.example.com/endpoint2'),
    fetch('https://api.example.com/endpoint3')
])
.then(responses => Promise.all(responses.map(res => res.json())))
.then(data => {
    console.log(data); // Array of results from all three APIs
})
.catch(error => console.error('Error:', error));
```

This approach ensures that all API requests are handled concurrently, and you can process the data once all of them have completed.

Q10: What are different Promise APIs available?
A10: JavaScript provides several Promise APIs:

- `Promise.all`: Resolves when all promises in an array resolve, or rejects if any promise rejects.
- `Promise.race`: Resolves or rejects as soon as the first promise in an array resolves or rejects.
- `Promise.allSettled`: Resolves when all promises have settled (either resolved or rejected), providing an array of objects describing the outcome of each promise.
- `Promise.any`: Resolves as soon as any one of the promises resolves, or rejects if all promises are rejected. These APIs offer flexibility in handling multiple asynchronous operations.

Q11: What are pure components?
A11: A pure component in React is a component that only re-renders when its props or state change. React's `PureComponent` implements `shouldComponentUpdate` with a shallow comparison of props and state, preventing unnecessary re-renders and improving performance. Pure components are ideal when a component renders the same output given the same props and state, making them more efficient in certain scenarios.

Q12: What is React and its advantage?
A12: React is a popular JavaScript library for building user interfaces, particularly single-page applications. It allows developers to create reusable UI components that manage their own state, leading to modular and maintainable code. React's virtual DOM optimizes rendering performance, and its component-based architecture simplifies the process of building complex UIs. Additionally, React's ecosystem supports server-side rendering, static site generation, and mobile app development with React Native.

Q13: Call API and print data using React
A13:

```
import React, { useState, useEffect } from 'react';

function App() {
    const [data, setData] = useState([]);

    useEffect(() => {
        fetch('https://jsonplaceholder.typicode.com/posts')
            .then(response => response.json())
            .then(data => setData(data))
```

```
            .catch(error => console.error('Error:', error));
    }, []);

    return (
        <div>
            <h1>Fetched Data</h1>
            <ul>
                {data.map(item => (
                    <li key={item.id}>{item.title}</li>
                ))}
            </ul>
        </div>
    );
}

export default App;
```

This code fetches data from an API and displays it in a list format.

Q14: Agile process followed in project
A14: The Agile process in a project involves iterative development, where requirements and solutions evolve through collaboration between cross-functional teams. Agile emphasizes flexible responses to change, continuous improvement, and delivering small, incremental updates rather than a single, large release. Scrum is a popular Agile framework, involving sprints (time-boxed iterations), daily stand-ups, sprint reviews, and retrospectives to ensure that the team stays on track and improves continuously.

Q15: Simple app using Redux
A15: A simple app using Redux might involve a counter with increment and decrement buttons:

1. Action: Define actions like `INCREMENT` and `DECREMENT`.
2. Reducer: Create a reducer function that updates the state based on the action type.
3. Store: Set up the Redux store using the reducer.
4. React Component: Connect the component to the Redux store using `connect` or `useSelector` and `useDispatch` hooks to access the state and dispatch actions.

Example:

```
// actions.js
export const increment = () => ({ type: 'INCREMENT' });
export const decrement = () => ({ type: 'DECREMENT' });

// reducer.js
const initialState = { count: 0 };
export function counterReducer(state = initialState, action) {
    switch (action.type) {
        case 'INCREMENT':
            return { ...state, count: state.count + 1 };
        case 'DECREMENT':
            return { ...state, count: state.count - 1 };
        default:
            return state;
    }
}

// store.js
import { createStore } from 'redux';
import { counterReducer } from './reducer';
const store = createStore(counterReducer);

// CounterComponent.js
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from './actions';

function CounterComponent() {
    const count = useSelector(state => state.count);
    const dispatch = useDispatch();

    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() =>
dispatch(increment())}>Increment</button>
            <button onClick={() =>
dispatch(decrement())}>Decrement</button>
```

```
        </div>
    );
}

export default CounterComponent;
```

Q16: Reverse given array
A16: Reversing an array in JavaScript can be done using the `reverse()` method:

```
const arr = [1, 2, 3, 4, 5];
const reversedArr = arr.reverse();
console.log(reversedArr); // Output: [5, 4, 3, 2, 1]
```

Alternatively, you can reverse an array manually using a loop or the spread operator and `reduce`:

```
const reversedArr = arr.reduce((acc, val) => [val, ...acc], []);
console.log(reversedArr); // Output: [5, 4, 3, 2, 1]
```

Q17: React lifecycle
A17: The React component lifecycle is divided into three phases: Mounting, Updating, and Unmounting. During mounting, components are created and inserted into the DOM. This phase involves methods like `constructor`, `componentDidMount`, and `render`. During updating, React re-renders components in response to changes in props or state, using methods like `shouldComponentUpdate`, `componentDidUpdate`, and `render`. During unmounting, components are removed from the DOM, with `componentWillUnmount` used to clean up resources.

Q18: Can you explain the difference between id and class attributes in HTML/CSS?
A18: In HTML and CSS, `id` and `class` attributes are used to identify elements and apply styles:

- `id`: Uniquely identifies a single element in the document. It should be used when you need to target exactly one element. An `id` must be unique within a page.

- `class`: Can be used to identify multiple elements with the same styling. It is useful for grouping elements that share the same style. Multiple elements can share the same `class` value.

Q19: How do you declare variables in JavaScript?
A19: Variables in JavaScript can be declared using `var`, `let`, or `const`:

- `var`: Declares a function-scoped or globally-scoped variable, depending on where it is declared. `var` allows re-declaration and has function-level scope.
- `let`: Declares a block-scoped variable, meaning the variable is only accessible within the block it is declared. `let` does not allow re-declaration in the same scope.
- `const`: Declares a block-scoped constant, which cannot be reassigned after its initial value is set. However, if the value is an object or array, its properties or elements can still be modified.

Q20: Can you explain what a function is and how you define one in JavaScript?
A20: A function in JavaScript is a reusable block of code that performs a specific task. Functions can take inputs (parameters) and return an output. Functions are defined using the `function` keyword, followed by a name, a list of parameters in parentheses, and a block of code in curly braces. For example:

```
function add(a, b) {
    return a + b;
}
```

This function `add` takes two parameters `a` and `b`, and returns their sum. Functions can be called (invoked) elsewhere in the code.