React Interview

useEffect

Some interview questions related to the `useEffect` hook:

1. Explain the `useEffect` hook. What arguments does it take?

2. What is the purpose of the `useEffect` hook in React?

   The primary purposes of the `useEffect` hook are:

   1. Executing Code After Render: It allows you to execute code after the component has rendered to the DOM.
   2. Managing Side Effects: You can use useEffect to manage and coordinate side effects within your component. This includes operations like fetching data from an API, updating the DOM, or interacting with external libraries.
   3. Dependency Tracking: `useEffect` can track dependencies, which are specified as an array of values in its second argument. When one or more of these dependencies change, the effect is re-executed. This allows you to respond to changes in state or props by running specific code.

3. What is the significance of the dependency array (`[]`) in `useEffect`? When is it used, and when is it omitted?

4. What happens if you provide an empty dependency array (`[]`) in `useEffect`?

5. How do you prevent `useEffect` from running on every render?

```
useEffect(() => {
  // This runs after every render
})

useEffect(() => {
  // This runs only on mount (when the component appears)
}, [])

useEffect(() => {
  // This runs on mount and also if either a or b have changed
}, [a, b])
```

COPY

6. How do you replicate the behaviour of `componentDidMount` using `useEffect`?

7. What are the differences between `useEffect` and `componentDidMount` in terms of lifecycle and usage?

   `useEffect` and `componentDidMount` serve similar purposes in React, but there are differences in terms of their lifecycle and usage due to the contrast between functional components (which use `useEffect`) and class-based components (which use `componentDidMount`).

   Lifecycle:

   1. Functional Components (`useEffect`): `useEffect` is a hook used in functional components. It is called after the initial render and also after each subsequent render.
   2. Class-Based Components (`componentDidMount`): `componentDidMount` is a lifecycle method available in class-based components. It is called only once, after the initial render.

   Cleanup:

   1. Functional Components (`useEffect`):

      - You can return a cleanup function from `useEffect` to handle resource cleanup when the component unmounts or when dependencies change before the next effect execution.

        💡 When a component unmounts, it means the component is being removed from the DOM, either because it's no longer needed or because the user navigated away from the page.

```
useEffect(() => {
  // Side effect logic

  return () => {
    // Cleanup logic
  }
}, [dependencyArray])
```

COPY

   2. Class-Based Components (`componentWillUnmount`):

      - Cleanup operations are typically performed in the `componentWillUnmount` lifecycle method in class-based components.

```
componentWillUnmount() {
  // Cleanup logic
}
```

COPY

   Both `useEffect` and `componentDidMount` allow you to handle side effects and perform actions after rendering, the key differences lie in the component types they are associated with, their lifecycle behaviours and the way cleanup is handled.

8. Explain the purpose of the cleanup function returned by `useEffect`. When and why is it used?

9. Explain how you can handle cleanup when unmounting a component using `useEffect`.

```
useEffect(() => {
  // Subscribe to an external service
  const subscription = subscribeToService()

  return () => {
    // Cleanup: Unsubscribe when component unmounts or when dependencies change
    subscription.unsubscribe()
  }
}, [dependency1, dependency2])
```

COPY

Cleanup is necessary to prevent memory leaks or resource leaks. It ensure that your component cleans up after itself when it's no longer in use or when its dependencies change.

10. How do you handle multiple `useEffect` calls within a single component?

    You can use multiple useEffects in a component.

```
useEffect(() => {
  // Effect 1: Handle one side effect
}, [dependency1])

useEffect(() => {
  // Effect 2: Handle another side effect
}, [dependency2])
```

COPY

Also it is better to group related logic together in the same `useEffect`. This can make your code more organised and easier to understand.

```
useEffect(() => {
  // Effect 1: Data fetching
  // ...
  // Effect 2: DOM manipulation
  // ...
}, [dependency1, dependency2])
```

COPY

useMemo

1. Explain the `useMemo` hook. What arguments does it take?

2. What is memoization, and how does it relate to useMemo?

    The `useMemo` hook in React is used for memoization, which means it memoizes (caches) the result of a function and returns that cached result whenever the dependencies specified in its dependency array remain unchanged.

```
const memoizedValue = useMemo(() => {
  // Memoized computation or value
  return result
}, [dependency1, dependency2])
```

COPY

Memoization Function: The first argument to `useMemo` is a function that contains the computation or value you want to memoize. This function is executed when one or more of the dependencies listed in the second argument change. The result of this function is memoized and returned.

Dependencies Array: The second argument is an array of dependencies. It contains values or references that the memoization function depends on. If any of these dependencies change between renders, the memoization function is re-executed, and the result is recalculated and memoized. If the dependencies remain the same, the cached result is returned.

Memoization is a technique used in programming to optimize and cache the results of expensive function calls so that the same result can be returned without re-computation, when the same input is encountered again. Memoization can significantly improve the performance of applications by reducing redundant calculations.

In React useMemo allows you to memoize values or computations based on dependencies, ensuring that the computed result is cached and reused when the dependencies remain the same between renders.

Code example:

```
import { useMemo } from 'react'

function MyComponent({ data }) {
  const computedValue = useMemo(() => {
    // Expensive computation based on 'data'
    return data.reduce((acc, val) => acc + val, 0)
  }, [data])

  return (
    <div>
      <p>Computed Value: {computedValue}</p>
    </div>
  )
}
```

COPY

3. How does `useMemo` help in optimizing the performance of a React application?

    1. One of the primary use cases for `useMemo` is to memoize (cache) the result of expensive computations. By memoizing values or computed results, you can avoid recomputing them in subsequent renders when the input data or dependencies remain unchanged. This reduces CPU and memory usage.
    2. Memoization helps optimize rendering performance by avoiding redundant work. When your component renders quickly, it contributes to a smoother and more responsive user experience.

4. An example of usage of useMemo.

A basic example is when filtering an array of data based on some conditions, such as applying a search filter to a list.

```
const filteredData = useMemo(() => {
  return data.filter((item) => item.name.includes(searchTerm))
}, [data, searchTerm])
```

COPY

5. What happens if you omit the dependencies array in `useMemo`?

   Without a dependencies array, `useMemo` does not track any dependencies. As a result, the memoized value is not recalculated when any values or references change in subsequent renders. It remains constant and does not respond to changes in props, state, or any other values.

   The memoized value won't update when the underlying data changes, and your component may not reflect the correct or updated state.

useCallback

useCallback is similar to useMemo in terms of syntax, difference is, it memoizes the callback function.

The `useCallback` hook serves the purpose of memoizing (caching) a callback function and preventing it from being recreated on every render of a component. It is particularly useful in scenarios where you want to optimize performance by avoiding unnecessary re-renders of child components that rely on callback functions as props.

When a component renders, JavaScript functions within it are recreated, including callback functions. If these callback functions are passed as props to child components, unnecessary re-renders of those child components can occur if the callbacks are recreated on each render of the parent component.

```
const memoizedCallback = useCallback(
  () => {
    // Callback function logic
  },
  [dependency1, dependency2, ...]
);
```

COPY

Reading reference : https://react.dev/reference/react/useCallback

useRef

Interview questions related to the `useRef` hook:

1. What is the use of the `useRef` hook in React?

    1. One of the primary use cases for `useRef` is to reference and interact with DOM elements in a React component. You can attach a ref to a DOM element, such as an input field or a div, and then access or manipulate that element's properties or values using the `current` property of the ref. This is useful for tasks like setting focus on an input field, scrolling to a specific location, or manually triggering certain DOM events.
    2. `useRef` can be used to store values across re-renders of a component without causing the component to re-render when the value changes. This makes it suitable for scenarios where you want to keep track of the previous state or props values.

2. How do you create a ref using the `useRef` hook in a functional component?

3. How can you access the current value of a ref created with `useRef`?

    https://codesandbox.io/s/muddy-shape-klhj46?file=/src/App.js

4. How does it differ from `useState`?

    - `useRef`: Changes to the value stored in a `useRef` do not trigger re-renders of the component. The component does not react to changes in the ref's value. It is primarily a way to store and access data without affecting rendering.
    - `useState`: Changes to the state variables created with `useState` trigger re-renders of the component. When the state changes, React automatically re-renders the component, updating the UI to reflect the new state.

    https://codesandbox.io/s/peaceful-diffie-vxpptv?file=/src/ExampleComponent.js

5. What is the difference between `useRef` and `createRef` when it comes to creating refs in React?

    `createRef` is used in class components. It works similar to `useRef`, but you can't directly provide an initial value when creating the ref.

    ```
    this.myRef = createRef() // Create a ref object
    ```

    COPY

Custom Hook

Let create a counter custom hook:

```
// App.js
import './styles.css'
import useCounter from './useCounter'

export default function App() {
  const { count, increment, decrement } = useCounter(0)
```

```
  return (
    <div className='App'>
      <h1>Counter</h1>
      <p>{count}</p>
      <button onClick={increment}>+</button>
      <button onClick={decrement}>-</button>
    </div>
  )
}
```

COPY

Now let's write the useCounter hook:

```
// useCounter.js
import { useState } from 'react'

export default function useCounter(initialValue) {
  const [count, setCount] = useState(initialValue)

  const increment = () => {
    setCount(count + 1)
  }

  const decrement = () => {
    setCount(count - 1)
  }

  return { count, increment, decrement }
}
```

COPY

What is a custom hook in React, and why would you use one?

Custom hooks are useful for reusing and sharing logic across different components. When creating custom hooks, it's essential to follow some rules and conventions to ensure they work correctly and are easy to understand and use. Here are some important points for creating custom hooks in React:

1. Prefix with "use": Custom hooks should always be named with a "use" prefix to signal that they follow the React hook conventions. For example, `useCounter`, `useFetchData`, or `useLocalStorage`.
2. Reusability: The primary goal of a custom hook is to encapsulate and abstract logic for reuse in multiple components.
3. Independent: Custom hooks should be independent of the specific components they are used in. Ensure that your hook provides a general-purpose solution that can be used in various contexts.
4. Arguments and Return Values: Custom hooks can accept arguments just like regular functions. They typically return values, often in the form of an array or object, to provide data and functions to components that use the hook.

Exercise:

Extract the custom Hook: https://codesandbox.io/s/pensive-archimedes-kmnzkf?file=/App.js

Practice Questions on Custom Hooks:

1. Create a custom hook called `useLocalStorage` that allows you to persist and retrieve data in local storage. It should have functions to set and get data.
2. You can find a lot of custom hooks on this website. Try to write the code on your own for them - https://usehooks.com/

Higher order components

They are functions that take a component as an argument and return a new component with additional props, state, or behavior. HOCs are a way to share behavior between components and promote code reuse. It is a concept primarily associated with class-based components in React.

Example:

```
import React from 'react'
import { withRouter } from 'react-router-dom'

function MyComponent(props) {
  const handleButtonClick = () => {
    // Programmatic navigation using history.push
    props.history.push('/new-page')
  }

  return (
    <div>
      <h1>My Component</h1>
      <button onClick={handleButtonClick}>Go to New Page</button>
    </div>
  )
}

export default withRouter(MyComponent)
```

COPY

About core of React

1. What is reconciliation in React?

2. Why is it essential for optimizing the rendering process?

3. Explain the virtual DOM and its role in reconciliation. How does it improve rendering performance in React?

4. What is the purpose of the `key` prop in React? How does it assist in reconciliation when rendering lists of components?

Reconciliation in React is the process of efficiently updating the user interface to reflect the most recent changes in a component's state or props. It is a core part of React's rendering process and is essential for optimizing the rendering of components. Here's why reconciliation is crucial for optimizing the rendering process in React:

1. Efficiency: Without reconciliation, React would need to re-render the entire component tree whenever there is any change in state or props. This would be highly inefficient and slow, especially for complex applications with many components.
2. Virtual DOM: React uses a virtual representation of the DOM (known as the virtual DOM) to perform reconciliation. Instead of directly updating the real DOM, React makes changes to the virtual DOM first. This allows React to batch updates, minimize the number of actual DOM manipulations, and optimize the rendering process.
3. Selective Updates: Reconciliation enables React to identify which parts of the component tree have changed and need to be updated. It compares the previous virtual DOM with the current one and calculates the minimum number of DOM operations required to bring the UI up to date. This selective update approach significantly improves performance.
4. List Rendering: When rendering lists of components, reconciliation ensures that React can efficiently add, remove, or reorder items in the list without re-rendering the entire list. The `key` prop plays a vital role in this process.
5. Components Reusability: Reconciliation ensures that components are re-rendered only when their state or props have actually changed. Components that don't need to be updated are not unnecessarily re-rendered, improving performance.
6. User Experience: Efficient reconciliation results in a smoother and more responsive user interface. Users perceive the application as faster and more pleasant to use when updates happen quickly and without noticeable delays.
7. Scalability: For large-scale applications, efficient reconciliation is essential to maintain a good user experience. It allows developers to build complex user interfaces without worrying too much about performance bottlenecks.

In summary, reconciliation in React is vital for optimizing the rendering process by minimizing the number of actual DOM updates, ensuring selective and efficient updates, and providing a smooth and responsive user experience. It is a fundamental aspect of React's performance optimization strategy.

💡 Checkout the guest lecture by Tanisha for visual understanding of React Internals.

5. What is diffing algorithm in React?

The "Diffing Algorithm" in React is the internal mechanism that React uses to efficiently update the user interface by calculating the differences (or "diffs") between the previous virtual DOM and the current virtual DOM.

Diffing Algorithm understanding

Reference Reading for React Hooks: https://react.dev/reference/react

Web Accessibility

Accessibility is a crucial aspect of web development that ensures that websites and web applications can be used by people with disabilities.

Here are some basic principles and tips for ensuring accessibility in your frontend development work:

1. Semantic HTML: Use proper HTML elements to structure your content. Semantic HTML elements like headings (`<h1>`, `<h2>`, etc.), lists (`<ul>`, `<ol>`), and form elements (`<form>`, `<input>`, `<label>`) provide context and structure for screen readers.

2. Keyboard Navigation: Ensure that all interactive elements on your website can be accessed and used with a keyboard alone. Test your website's tab order and make sure it follows a logical flow.

3. Focus Styles: Provide clear and visible focus styles for interactive elements (e.g., links and buttons) so that keyboard users can easily see where they are on the page.

4. Alt Text: Always include descriptive and meaningful alternative text for images (`<img>`). This is essential for users who are blind or have low vision.
```
<img src="apple.jpg" alt="A red apple on a wooden table" />
```
COPY

5. Labels for Form Controls: Use `<label>` elements to associate text labels with form controls. This helps screen reader users understand the purpose of each field.
```
<label for="username">Username:</label>
<input type="text" id="username" name="username" />
```
COPY

6. Use ARIA: ARIA (Accessible Rich Internet Applications) attributes can enhance the accessibility of complex UI components like sliders, modals, and accordions.

   https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Attributes

   https://web.dev/learn/accessibility/aria-html/

7. Color Contrast: Ensure sufficient color contrast between text and background to make content readable for users with low vision or color blindness. Tools like the WebAIM Color Contrast Checker can help you assess this.

   https://webaim.org/resources/contrastchecker/

8. Semantic Headings: Use headings (`<h1>`, `<h2>`, etc.) to create a hierarchical structure for your content. Headings should reflect the logical organization of the page.

9. Captions and Transcripts: Provide captions or transcripts for audio and video content. This benefits users who are deaf or hard of hearing.

10. Testing Tools: Use accessibility testing tools such as Lighthouse to automatically check for common accessibility issues and get suggestions for improvements.

11. Avoid Auto-Playing Media: Don't auto-play audio or video content. If it's necessary, provide controls for users to pause or mute it.

12. Error Handling: Provide clear error messages and suggestions for resolving form errors. Avoid using color alone to indicate errors.