Crush Your React Interview: A Last Minute Preparation Guide

Have only a week before your next react interview? Or maybe just 2 days?

Well then, Get ready for your React JS interview in no time with this comprehensive last minute preparation guide!

This handbook covers all the essential React JS interview questions and provides answers, tips and explanations to help you confidently tackle any interview situation.

Don't let stress and lack of preparation hold you back, grab your copy now and be fully prepared for your next big opportunity!

For free video tutorials, checkout and subscribe to xplodivity on youtube.

Content list:

8. What is the difference between react.fragment and <></> in react?
9. What is Profiler in react?
10. What is suspense in react?
11. What is StrictMode?
12. How to simulate lifecycle methods in react hooks?
13. What is a pure component?
14. What is forward ref in react hooks? Explain with example.
15. How to send data from child to parent component in react?
16. What is redux? Why is it important?
17. What is the difference between useContext and redux, When to use either, and why?
18. When to use useReducer in components?
19. What are some issues in using the useContext hook?
20. How to use useReducer and useContext together?
21. Can you initialize useState with a function?
    OR
    How to perform Lazy Initialization of the State?
22. How to manipulate dom with refs?
23. What are keys in react and how do they improve performance?
24. Is it recommended to use index as a key in react? Explain.
25. What is the difference between controlled and uncontrolled components?
26. What does one-way data binding mean in react?
27. What are some limitations of react?
28. What is a higher order component in react?
29. when to use class over functional component in react or vice versa?
30. What are stateless and stateful components?
31. What is the cleanup function in useEffect? Why is it used? How does it help?
32. What is a memory leak in react? Explain with example.
33. What is the cleanup function in useEffect? Why is it used? How does it help?
34. What is lazy loading (code splitting) in react?
35. Talk about some commonly used custom hooks in react.
    - useWindowSize hook
    - useStorage hook
    - useDebounce hook
36. What are the different ways a component can be re-rendered in react?

Note- This is an interview preparation guide and therefore the list contains questions of basic, intermediary, and advanced levels arranged in random order. You can start with any question in any order.

1- what is virtual dom in react?

Ans- The Virtual DOM is an abstract representation of the actual DOM, and is used to increase the performance of updates and rendering of the UI. When a component state changes, React updates the Virtual DOM, which then calculates the difference between the previous and current version of the Virtual DOM. React then updates only the parts of the actual DOM that have changed, instead of re-rendering the entire tree, resulting in improved performance.

Virtual DOM:

```
{
  type: 'div',
  props: {
    className: 'container',
    children: [
      { type: 'h1', props: { children: 'Hello, World!' } },
      { type: 'p', props: { children: 'This is a virtual DOM' } }
    ]
  }
}
```

Real DOM:

```
<div class="container">
  <h1>Hello, World!</h1>
  <p>This is a virtual DOM</p>
</div>
```

2- Is react a UI library or framework? and why?

Ans- React is considered a library rather than a framework because it focuses solely on the view layer of an application. It provides a set of tools and functionalities specifically for building user interfaces, but leaves the decision on how to manage

state, structure the overall application, and handle data flow to the developer. In contrast, a framework defines a set of conventions and guidelines for building an entire application and often includes an opinionated structure and way of handling such tasks. So, React provides a flexible and modular approach to UI development, making it a library.

3- what is jsx?

Ans- JSX stands for JavaScript XML. It allows us to write HTML in react in an easier way.
With JSX, you can write syntax that looks like HTML, but instead of being interpreted as static HTML, it is transformed into JavaScript functions that return elements and components. This provides a way to define the structure and appearance of user interfaces using a syntax that is easy to understand and familiar for many developers.

For example, this is JSX code:

```
const element = <h1>Hello, World!</h1>;
```

The above JSX code gets transformed into the following JavaScript:

```
const element = React.createElement("h1", null, "Hello, World!");
```

As you can see, JSX allows us to write HTML elements in JavaScript and place them in the DOM without the use of createElement(), making the code much simpler.

Some more examples of jsx:

- you can write expressions inside curly braces {} in the following way:

```
const myElement = <h1>React is {5 + 5} times better with JSX</h1>;
```

- To write HTML on multiple lines, put the block of HTML within parentheses:

```
const myElement = (
  <ul>
    <li>Car</li>
    <li>Smartphone</li>
```

```
  </ul>
);
```

4- why prefer react over angular or vue?

Ans- React, Angular, and Vue are all popular JavaScript frameworks for building web applications, and each has its own benefits and drawbacks. Here's a more in-depth comparison of React with Angular and Vue:

1. **Performance:** React is known for its fast performance, thanks to its virtual DOM, which allows it to update the UI efficiently without having to touch the entire DOM. In comparison, Angular has a slightly slower performance, as it uses two-way data binding, which can result in slow updates and render times. Vue also uses a virtual DOM, but its performance is not quite as fast as React's, although it is still considered fast compared to other frameworks.
2. **Learning Curve:** React has a relatively shallow learning curve, as it uses a simple API and focuses on components. This makes it easier to learn, especially for developers who are already familiar with JavaScript. Angular, on the other hand, has a steeper learning curve, as it uses a complex architecture, multiple libraries, and TypeScript, which can be difficult for new developers to understand. Vue has a similar learning curve to React, as it also focuses on components and has a simple API, but it is not quite as intuitive as React.
3. **Community:** React has a large and active community, which means that it has a wealth of resources, tutorials, and libraries available. Angular also has a large community, although it is not quite as active as React's. Vue has a growing community, but it is still smaller compared to the other two frameworks.
4. **Flexibility:** React is considered to be more flexible than Angular, as it gives developers more control over the structure of their code and does not dictate how applications should be built. Angular, on the other hand, is more opinionated and requires developers to follow a specific structure and architecture. Vue is similar to React in terms of flexibility, but it is not quite as customizable as React.
5. **Server-side Rendering:** React has strong support for server-side rendering, which can improve the performance of your application, especially for users with slow internet connections. Angular also supports server-side rendering, although it is not quite as fast as React's. Vue also supports server-side rendering, but it is not as well-developed as React's or Angular's.

Overall, React is a great choice for developers who want a fast, flexible, and easy-to-learn framework. Angular is best for developers who want a more

opinionated framework with a strong structure and architecture. Vue is a good choice for developers who want a flexible and easy-to-learn framework that is similar to React, but with a smaller community and slightly slower performance.

5- How does rendering work in react (describe the lifecycle of a functional component only) ?

Ans- A React component undergoes three phases in its lifecycle: mounting, updating, and unmounting.

1. **Mounting phase:** when a new component is created and inserted into the DOM or, in other words, when the life of a component begins. This can only happen once, and is often called “initial render.”
2. **Updating phase:** is when the component updates or re-renders. This reaction is triggered when the props are updated or when the state is updated. This phase can occur multiple times, which is kind of the point of React.
3. **Unmounting phase**: The last phase within a component's lifecycle, when the component is removed from the DOM.

The rendering process of a React functional component is as follows:

1. **Initial render:** When a functional component is first rendered, React creates a virtual DOM and updates it with the component's initial state and props.
2. **Virtual DOM comparison:** React compares the updated virtual DOM with the previous virtual DOM to determine what changes have been made.
3. **DOM updates:** React makes the necessary updates to the real DOM based on the comparison of the virtual DOM. This includes adding, removing, or updating elements as needed.
4. **State and prop updates:** If there is a change in the state or props of the component, React will repeat the virtual DOM comparison and DOM update process.
5. **Rendering with useEffect:** If the component uses the useEffect hook, React will call the useEffect callback after the initial render and whenever the specified values change. This allows you to perform side-effects such as making API calls or updating the component state.
6. **Cleanup with useEffect:** If the useEffect callback returns a function, React will call this function before unmounting the component or before running the next useEffect call. This function can be used for cleanup tasks, such as removing event listeners.
7. **Final render:** Once the DOM updates have been completed, the component will be fully rendered and ready to interact with the user.

This process will repeat each time the component's state or props change, or whenever the useEffect hook is triggered, ensuring that the component always stays up-to-date with the latest data.

6- Talk about some common important react hooks

Ans- some common important react hooks are - useState, useRef, useMemo, useCallback, useEffect, useReducer, useContext.

useState: useState is a React Hook that allows you to manage state in your functional components. It's a way to store data that changes over time, such as user input or the response from a network request.
Here's an example of how to use the useState hook in a React component:

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

const ExampleComponent = () => {
  // Declare a state variable named "count" and initialize it to 0
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
};

export default ExampleComponent;
```

In this example, we use the useState hook to declare a state variable named count and initialize it to 0. The useState hook returns an array with two elements: the current value of the state (count) and a setter function (setCount) that allows you to update the state.
In the component's render method, we display the current value of the count state and attach a onClick handler to the button that increments the count value by 1 each time the button is clicked. When the count value changes, the component will re-render with the updated value.

Potential follow-up question by interviewer -

what is the callback in setState used for in react?

In React, the setState method is used to update the state of a component. The second argument to setState is an optional callback function that will be executed after the state has been updated.
This callback function is useful when you need to perform some action after the state has been updated. For example, you may want to update the UI based on the new state or make an API request that depends on the updated state.
Here is an example:

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

function ExampleComponent() {
  const [count, setCount] = useState(0);

  const handleClick = () => {
    setCount(prevCount => prevCount + 1, () => {
      console.log('Count has been updated to', count);
    });
  };

  return (
    <div>
      <p>{count}</p>
      <button onClick={handleClick}>Increase Count</button>
    </div>
  );
}
```

useEffect: useEffect is a React Hook that lets you synchronize a component with an external system. For example, you can use it to automatically fetch data or update the title of a document when a component mounts or updates.
The useEffect hook takes two arguments:

1. A callback function that contains the "effect logic"
2. An array of dependencies (optional).

The effect logic is executed after rendering the component and any updates to it. If the dependencies array is omitted or empty, the effect will run on every render. If it includes values, the effect will only run if one of the values changes.

Potential follow-up question by interviewer -

what is a side effect in react?

Ans- In React, a "side effect" refers to an operation that can affect something outside of a component's render function, such as a network request, setting a timer, or changing the DOM in some way. Side effects can make a component's behavior unpredictable and harder to understand, so it's best to manage them carefully.
The useEffect hook allows you to perform side effects in your functional components. By declaring your side effects with useEffect, you ensure that they are only run when they need to be run, and not on every render.
For example, you might want to make a network request when a component mounts, and clean it up when the component unmounts. Here's how you would do that using useEffect:

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

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

  useEffect(() => {
    fetch('https://my-api.com/data')
      .then(response => response.json())
      .then(data => setData(data));

    return () => {
      // Clean up the effect.
    };
  }, []);

  return data === null ? 'Loading...' : <div>{data.message}</div>;
}
```

Here are four different examples of using useEffect in React:

1. Basic example:

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

function ExampleComponent() {
  useEffect(() => {
```

```
    console.log('Component has mounted');
  }, []);

  return <div>Hello World</div>;
}
```

2. Example with cleanup:

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

function ExampleComponent() {
  useEffect(() => {
    console.log('Component has mounted');

    return () => {
      console.log('Component will unmount');
    };
  }, []);

  return <div>Hello World</div>;
}
```

3. Example with Dependency Array:

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

function ExampleComponent() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log(`Count has changed to ${count}`);
  }, [count]);

  return (
    <div>
      <p>{count}</p>
      <button onClick={() => setCount(count + 1)}>Increase
Count</button>
    </div>
```

```
  );
}
```

4. Example with Asynchronous Code:

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

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

  useEffect(() => {
    async function fetchData() {
      const response = await
fetch('https://api.example.com/data');
      const json = await response.json();

      setData(json);
    }

    fetchData();
  }, []);

  return <div>{data ? JSON.stringify(data) : 'Loading...'}</div>;
}
```

useRef: useRef is a hook in React that allows you to store a value that won't change, even if the component re-renders. It's often used to store a reference to a DOM element or a value that needs to persist across render cycles.
useRef returns a mutable object with a property current which can hold a reference to a value. The value stored in current persists across render cycles, making it useful for storing references to elements, timer IDs, or other non-reactive values that should persist throughout the component's lifecycle.

Here's an example of using the useRef hook to store a reference to an input element in a form and later accessing its value in the component's submit handler:

```
import React, { useRef } from "react";

function FormExample() {
```

```
  const inputRef = useRef(null);

  const handleSubmit = (event) => {
    event.preventDefault();
    console.log(inputRef.current.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={inputRef} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

In this example, useRef returns an object with a current property that's initialized to null. We then pass the inputRef object as the ref attribute of the input element. In the submit handler function handleSubmit, we can access the value of the input by calling inputRef.current.value.

useMemo: useMemo is a React hook that allows you to optimize your component performance by only re-computing the value when a specific condition changes. It is used to store and return a memoized value (i.e. a cached value that is remembered across re-renders) that is expensive to calculate. The idea behind useMemo is to avoid performing unnecessary work and to improve performance by only recalculating the value when necessary.

Here's an example of how you can use the useMemo hook in React to memoize a expensive calculation:

```
import React, { useMemo } from 'react';

const ExpensiveCalculation = ({ data }) => {
  const memoizedValue = useMemo(() => {
    let sum = 0;
    for (let i = 0; i < data.length; i++) {
      sum += data[i];
    }
    return sum;
  }, [data]);

  return <div>{memoizedValue}</div>;
```

```
};
```

In this example, useMemo is used to memoize the result of the expensive calculation sum. The second argument to useMemo is a dependency array, which tells React when to re-compute the memoized value. In this case, the memoized value will only be recomputed if data changes.

useCallback: useCallback is a React Hook that returns a memoized version of a callback function. The returned function is guaranteed to have the same reference for as long as the dependencies remain the same, allowing React to skip unnecessary re-renders. The Hook takes two arguments: the callback function and an array of dependencies. The dependencies are values from your component state or props that the callback function depends on. When any of the dependencies change, the memoized callback will be updated with the new values.

For example

```
const MyComponent = ({ onClick, data }) => {
  const memoizedCallback = useCallback(() => {
    console.log(data);
  }, [data]);

  return <button onClick={memoizedCallback}>Click me</button>;
};
```

In this example, memoizedCallback will only change when data changes. This can be useful in performance optimization, especially when dealing with child components that re-render frequently.

IMPORTANT NOTE- When functions are passed as a prop to a child component from the parent component, whenever the parent component re-renders due to state change, the child component also re-renders because functions are recreated during a component rerender, so a new function is passed down to the child component during every parent re-render which causes the child component to rerender as well. But using useCallback, a new function will not be passed to the child component during the parent component rerender because The returned function in useCallback is guaranteed to have the same reference for as long as the dependencies remain the same, allowing React to skip unnecessary re-renders.

useReducer: useReducer is a hook in React that allows managing state in a functional component. It's similar to useState, but is more suitable for complex state management.

It takes two arguments:

- reducer: a pure function that takes the previous state and an action and returns the next state.
- initialState: the initial state of the component.

It returns an array with two values:

- The current state, updated by the reducer function.
- A dispatch function that can be called to trigger the reducer and update the state.

The useReducer hook can be used to manage complex state transitions that would require multiple useState hooks or if you have a global state that multiple components depend on.

Here is an example of how you might use the useReducer hook to manage a simple counter state:

```
import React, { useReducer } from 'react';

const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type:
'increment'})}>+</button>
      <button onClick={() => dispatch({type:
```

```
'decrement'})}>-</button>
    </>
  );
}
```

In this example, the useReducer hook is used to manage the state of a counter, which is stored in the state object. The dispatch function is used to trigger updates to the state by passing an action with a type of "increment" or "decrement". The reducer function is then used to update the state based on the type of action.

useContext: The useContext hook in React allows you to access data stored in a Context object in your functional components. Context provides a way to pass data through the component tree without having to pass props down manually at every level.
Using useContext is straightforward. First, you need to create a Context object using the React.createContext method and then pass in a default value. Then, you can use the useContext hook inside your functional component to access the data stored in the Context object.

Here's a simple example:

```
import React, { createContext, useContext } from 'react';

const MyContext = createContext({});

function ComponentA() {
  const context = useContext(MyContext);
  return (
    <div>
      {context.message}
    </div>
  );
}

function App() {
  return (
    <MyContext.Provider value={{ message: 'Hello from Context' }}>
      <ComponentA />
    </MyContext.Provider>
  );
}
```

In this example, MyContext is a Context object that is initialized with an empty object as its default value. ComponentA is a functional component that uses useContext to access the value stored in MyContext. The App component is a top-level component that wraps ComponentA in a MyContext.Provider component, providing the value to be accessed by useContext.

7- what is a fragment in react and why is it used?

Ans- In React, a Fragment is a way to group a list of children without adding an extra node to the DOM (Document Object Model). It allows you to return multiple elements from a component's render method without wrapping them in an extra div.

Fragments are used to improve code readability, reduce the amount of unnecessary markup, and to resolve issues related to rendering a list of children elements. It is also used to group elements that don't have a semantic meaning on their own but need to appear together.

Fragments are declared using the syntax <React.Fragment> or the short-hand <> syntax.

For example:

```
render() {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  );
}
```

Or using the short-hand syntax:

```
render() {
  return (
    <>
      <ChildA />
      <ChildB />
      <ChildC />
```

```
    </>
  );
}
```

8- what is the difference between react.fragment and <></> in react?

Ans- The only difference between them is that the shorthand version (<></>) does not support the key attribute.

React.Fragment supports key attribute, example:

```
  {
    array.map(arr => (
       <React.Fragment key={arr}>
         <p>The next array is...</p>
         <p>{fruit}!</p>
       </React.Fragment>
     ))
  }
```

9- What is Profiler in react?

Ans- React Profiler is a performance tool in React that helps you to analyze and optimize the rendering performance of your components. It provides information about how long it takes for components to render and update.
Here's an example of how to use the React Profiler in a component:

```
import React, { Profiler } from 'react';

function MyComponent() {
  return (
    <Profiler id="MyComponent" onRender={(id, phase,
actualDuration) => {
      console.log(id, phase, actualDuration)
    }}>
```

```
      {/* your component content */}
    </Profiler>
  );
}
```

In this example, the Profiler component is used to wrap MyComponent. The id prop is used to identify the component being profiled, and the onRender prop is a callback function that gets called every time the component renders. The onRender callback receives three arguments: id, phase, and actualDuration.

id is the identifier for the component being profiled, phase can be either mount (when the component is first added to the DOM) or update (when the component updates), and actualDuration is the time it took for the component to render in milliseconds.

By using the React Profiler, you can get insights into the performance of your components and make improvements to optimize the overall performance of your React application.

10- What is suspense in react?

Ans- React Suspense is a feature in React that allows you to handle loading state for a component asynchronously. It provides a way to delay rendering of a component until some data required for rendering is available.

For example, consider a component that fetches data from an API and displays it:

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

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

  useEffect(() => {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => setData(data))
  }, []);

  if (!data) {
    return <div>Loading...</div>;
  }
```

```
  return (
    <ul>
      {data.map(item => (
        <li key={item.id}>{item.title}</li>
      ))}
    </ul>
  );
}
```

In this example, the component first shows a loading message while the data is being fetched, and then renders the list of items once the data is available. With Suspense, you can handle this loading state more declaratively:

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

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

  useEffect(() => {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => setData(data))
  }, []);

  return (
    <Suspense fallback={<div>Loading...</div>}>
      <ul>
        {data.map(item => (
          <li key={item.id}>{item.title}</li>
        ))}
      </ul>
    </Suspense>
  );
}
```

In this example, the Suspense component is used to wrap the component that requires data. The fallback prop specifies what to render while the data is being fetched. The component inside the Suspense component can be rendered as soon as the data is available, without having to manage the loading state manually.

React Suspense is useful for improving the user experience and avoiding the "flash

of loading content" that can occur when loading data asynchronously.

11- What is StrictMode?

Ans- React StrictMode is a feature in React that helps you to identify potential problems in your code during development. It's a testing tool that runs extra checks and warnings for your components and helps you to write high-quality code.

Here's an example of how to use StrictMode in a React application:

```
import React, { StrictMode } from 'react';
import App from './App';

function MyApp() {
  return (
    <StrictMode>
      <App />
    </StrictMode>
  );
}

export default MyApp;
```

In this example, the StrictMode component is used to wrap the root component of the application. StrictMode runs additional checks and warnings for components within its scope and can help you to identify potential problems in your code.
For example, StrictMode can help you to detect problems such as:

- Components that render unnecessarily
- Use of legacy methods or APIs
- Potential performance bottlenecks

Using StrictMode during development can help you to write better, more optimized code and to avoid common mistakes that can negatively impact performance. Note that StrictMode only runs in development mode, and its warnings and checks do not affect the performance of your application in production.

12- How to simulate lifecycle methods in react hooks?

Ans- In React Hooks, you can simulate lifecycle methods by using a combination of React Hooks, such as useEffect and useState.

Here are some examples of how to simulate lifecycle methods with React Hooks:

componentDidMount: You can use the useEffect hook with an empty dependency array to simulate componentDidMount. The useEffect hook will run only once, after the first render:

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

function MyComponent() {
  useEffect(() => {
    // componentDidMount logic
    console.log('Mounted');
  }, []);

  return (
    <div>My component</div>
  );
}
```

componentDidUpdate: You can use the useEffect hook with dependencies to simulate componentDidUpdate. The useEffect hook will run after every render and will compare the dependencies from the previous render to the current render. If the dependencies have changed, the effect will run:

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

function MyComponent({ data }) {
  const [count, setCount] = useState(0);

  useEffect(() => {
    // componentDidUpdate logic
    console.log('Updated');
  }, [count, data]);

  return (
    <div>
      My component
```

```
      <button onClick={() => setCount(count + 1)}>Click
me</button>
    </div>
  );
}
```

componentWillUnmount: You can use the useEffect hook with a cleanup function to simulate componentWillUnmount. The cleanup function is run before the component is unmounted and can be used to perform any necessary cleanup:

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

function MyComponent() {
  const [isMounted, setIsMounted] = useState(true);

  useEffect(() => {
    // componentDidMount logic
    console.log('Mounted');

    // componentWillUnmount logic
    return () => {
      console.log('Unmounted');
    };
  }, []);

  return isMounted ? (
    <div>
      My component
      <button onClick={() => setIsMounted(false)}>Unmount</button>
    </div>
  ) : null;
}
```

These are just examples and you can use React Hooks to simulate other lifecycle methods as well. The goal is to provide a way to implement the logic of lifecycle methods in a more functional and composable way.

13- what is a pure component?

Ans- A "Pure Component" in React is a component that implements the shouldComponentUpdate lifecycle method with a shallow comparison of the current and next props and state. This method determines whether the component should re-render or not.

In practice, a Pure Component is a component that only re-renders when its props or state have changed. This can help to improve performance by avoiding unnecessary re-renders, as well as making the component behavior more predictable.

Here's an example of a Pure Component in React:

```
import React, { PureComponent } from 'react';

class MyComponent extends PureComponent {
  render() {
    return (
      <div>{this.props.value}</div>
    );
  }
}
```

In this example, the MyComponent component extends PureComponent and implements the render method. Because MyComponent extends PureComponent, it automatically implements the shouldComponentUpdate method with a shallow comparison of the current and next props and state. This means that the component will only re-render if the value of its props has changed.

14- What is forward ref in react hooks? Explain with example.

Ans- "Forward Refs" in React Hooks are a way to pass a reference from a parent component to a child component, allowing the parent component to access the child component's properties or methods. This can be useful in situations where the child component is a stateless functional component and doesn't have access to ref props.

Forward refs in React Hooks are created using the forwardRef function and the useImperativeHandle hook. The forwardRef function is used to wrap the child component and accept a ref parameter. The useImperativeHandle hook is used to pass a custom object through the ref, which the parent component can use to access the child component's properties or methods.

Here's an example of how to use forward refs in React Hooks:

```
import React, { forwardRef, useImperativeHandle } from 'react';

const ChildComponent = forwardRef((props, ref) => {
  const inputRef = useRef(null);

  useImperativeHandle(ref, () => ({
    focus: () => inputRef.current.focus()
  }));

  return <input ref={inputRef} type="text" />;
});

const ParentComponent = () => {
  const inputRef = useRef(null);

  const handleClick = () => {
    inputRef.current.focus();
  };

  return (
    <>
      <ChildComponent ref={inputRef} />
      <button onClick={handleClick}>Focus Input</button>
    </>
  );
};
```

In this example, the ChildComponent is wrapped with forwardRef and receives a ref parameter. The useImperativeHandle hook is used to pass a custom object through the ref, which contains a focus method that focuses the input element. In the parent component, a ref is created with useRef and passed to the child component as a prop. When the button is clicked, the input is focused by calling inputRef.current.focus().

15- how to send data from child to parent component in react?

Ans- In React, you can send data from a child component to a parent component by

passing a callback function as a prop to the child component. The child component then calls this callback function and passes the data as an argument.

Here's an example:

```
// Parent component
function ParentComponent(props) {
  const handleChildData = (data) => {
    console.log(data);
  };

  return (
    <div>
      <ChildComponent sendData={handleChildData} />
    </div>
  );
}

// Child component
function ChildComponent(props) {
  const sendData = () => {
    props.sendData('Data from child');
  };

  return (
    <button onClick={sendData}>Send Data</button>
  );
}
```

In this example, the ParentComponent passes a callback function handleChildData as a prop to the ChildComponent. The ChildComponent then calls this function and passes the data 'Data from child' when a button is clicked.

16- What is redux? Why is it important?

Ans- Redux is a state management library commonly used with React.js, it helps in managing the application's global state and enables data flow in a single direction, making it easier to understand and debug the application.
Redux is important in React because it helps to address some of the limitations of the React component state, such as the difficulty of sharing data between components and managing complex state updates. With Redux, the state of the whole application is stored in a single object called the store, and the components

can access the state by using selectors, without modifying it directly. This makes it easier to maintain consistency and avoid potential bugs in large-scale React applications.
Key points in using redux:

- React Components: React components make up the UI of the application.
- Redux Store: This is a single object that holds the entire state of the application.
- Actions: Actions are objects that describe changes to the state. They are dispatched by the React components.
- Reducers: Reducers are functions that handle the actions and update the state accordingly. They are responsible for updating the state object in the store.
- State Object: This is the data that describes the current state of the application. The state is updated by the reducers in response to actions.

Here is a simple code example that illustrates how to use Redux in React:

```
// Actions
const ADD_TODO = 'ADD_TODO';
const REMOVE_TODO = 'REMOVE_TODO';

// Action Creators
function addTodo(text) {
  return { type: ADD_TODO, text };
}

function removeTodo(index) {
  return { type: REMOVE_TODO, index };
}

// Reducer
function todosReducer(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [...state, action.text];
    case REMOVE_TODO:
      return state.filter((_, i) => i !== action.index);
```

```
    default:
      return state;
  }
}

// Store
import { createStore } from 'redux';
const store = createStore(todosReducer);

// React Component
import React, { useState } from 'react';
import { useSelector, useDispatch } from 'react-redux';

function TodoApp() {
  const [text, setText] = useState('');
  const todos = useSelector(state => state);
  const dispatch = useDispatch();

  function handleSubmit(event) {
    event.preventDefault();
    dispatch(addTodo(text));
    setText('');
  }

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <input type="text" value={text} onChange={e =>
setText(e.target.value)} />
        <button type="submit">Add Todo</button>
      </form>
      <ul>
        {todos.map((todo, index) => (
          <li key={index}>
            {todo}
            <button type="button" onClick={() =>
dispatch(removeTodo(index))}>
              Remove
            </button>
          </li>
        ))}
      </ul>
```

```
    </div>
  );
}
```

In this example, we first define the ADD_TODO and REMOVE_TODO actions, and the corresponding action creators addTodo and removeTodo.

Next, we create a todosReducer function that updates the state of the todos list in response to actions. The state of the todos list is represented as an array of strings.

We then create a Redux store using the createStore function and passing in the todosReducer as the only argument.

Finally, we create a React component called TodoApp that uses the useSelector and useDispatch hooks from the react-redux library to access the state of the todos list and dispatch actions to the store. The component allows the user to add and remove todos.

This is just a simple example, but it should give you an idea of how Redux can be used in React to manage the state of an application.

17- What is the difference between useContext and redux, When to use either, and why?

Ans- useContext and Redux are two different approaches for managing the state of a React application.

useContext is a hook that provides a way to share data between components without passing props down the component tree manually. It's a simpler approach compared to Redux and is often used for sharing global data that doesn't need to be updated frequently, such as themes or translations.

Redux, on the other hand, is a state management library that provides a centralized store for the entire application state and a way to update the state with actions and reducers. It's often used for more complex state management where the state needs to be updated frequently and in a predictable way, such as in a data-driven application.

As for which one to prefer, it depends on the specific requirements of your application. If your state management needs are simple, useContext might be sufficient. But if your state management needs are more complex, Redux provides a

more robust solution.

In general, if you are building a large or complex application with many interrelated state updates, or if you need to manage state in a way that can be easily tested and debugged, Redux might be the better choice. On the other hand, if you are building a simple or small application, or if you just need to share a small amount of global data between components, useContext might be a simpler and more lightweight solution.

18- when to use useReducer in components?

Ans- useReducer is a hook that provides an alternative to using useState for managing state in a React component. You can use useReducer when you have state updates that are complex, have multiple related updates, or have updates that depend on previous state.

For example, consider a form with multiple fields, each with its own state. The state updates for one field might depend on the state of another field, so you would need to write custom logic to handle the updates. In such a case, useReducer would be more appropriate than useState.

Another use case for useReducer is when you need to perform complex updates to the state. For example, if you need to calculate the next state based on the current state, or if you need to update multiple pieces of state at once, useReducer can make your code more maintainable and easier to understand.

So, in general, you should use useReducer when:

- You need to handle complex state updates
- You have multiple related updates to the state
- Your updates depend on previous state

And, you should use useState when:

- You have simple state updates
- Your updates don't depend on previous state
- You only need to update a single piece of state.

Note- A good example of using useReducer would be the implementation of a cart system.

19- what are some issues in using the useContext hook?

Ans- Some of the issues in using the useContext hook are:

Performance: The useContext hook can create performance issues because it causes components to re-render whenever the context value changes. If the context value is updated frequently, this can lead to excessive re-rendering and potential performance issues.

Re-rendering a component is an expensive operation that can impact the performance of an application, especially if there are many components that are subscribed to the same context. To reduce this, it is important to minimize the number of context updates and to make sure that they are only performed when necessary.

Implicit relationships: The useContext hook creates implicit relationships between components by allowing components to subscribe to a context and re-render whenever the context value changes. This means that if a component is using useContext to subscribe to a context, it is dependent on that context for its state and behavior.

These implicit relationships can make it difficult to understand how changes to the context will affect the components using it. For example, if a context value is updated in one part of the application, it can have unintended consequences for components that are subscribed to that context elsewhere in the application.

This can make it difficult to understand the full impact of a change to the context, especially if the relationships between components using the context are complex. To mitigate this, it is important to design context carefully and to understand the relationships between components that are using it.

20- How to use useReducer and useContext together?

Ans- Here's an example of how to use useReducer and useContext together:

```
import React, { useReducer, useContext } from 'react';

const initialState = { count: 0 };

const reducer = (state, action) => {
  switch (action.type) {
    case 'increment':
```

```
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
};

const CountContext = React.createContext();

const CountProvider = ({ children }) => {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <CountContext.Provider value={{ state, dispatch }}>
      {children}
    </CountContext.Provider>
  );
};

const Counter = () => {
  const { state, dispatch } = useContext(CountContext);
  return (
    <>
      <p>{state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>
       +
      </button>
      <button onClick={() => dispatch({ type: 'decrement' })}>
       -
      </button>
    </>
  );
};

const App = () => {
  return (
    <CountProvider>
      <Counter />
    </CountProvider>
  );
};
```

```
export default App;
```

In the example, we start by creating a reducer function that takes the current state and an action as arguments and returns a new state based on the action type. The reducer is responsible for updating the state based on the actions it receives.

Next, we create a context using React.createContext(). The context will allow us to share state and dispatch function between components.

Then we create a CountProvider component that uses the useReducer hook to manage the state. The useReducer hook takes two arguments: the reducer function and an initial state. It returns an array containing the current state and a dispatch function that can be used to dispatch actions to the reducer. The CountProvider component wraps its children with the CountContext.Provider component and passes down the state and dispatch function as context values.

Finally, we create a Counter component that uses the useContext hook to access the state and dispatch function from the context. The useContext hook takes the context object as an argument and returns the current context value, which is an object with the state and dispatch function in this case. In the Counter component, we display the current count and two buttons that dispatch increment and decrement actions when clicked.

With this setup, we can use the CountProvider component to wrap any components that need access to the count state and dispatch function. The state and dispatch function will be passed down the component tree via context, allowing the child components to access and update the state without having to pass props down manually.

21- Can you initialize useState with a function?

OR

How to perform Lazy Initialization of the State?

Ans- Yes, you can initialize useState with a function. The function is executed only once when the component is mounted, and it returns the initial state value.

Here's an example:

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

const Example = () => {
  const [count, setCount] = useState(() => {
    // Calculate the initial state value
    const initialValue = someExpensiveComputation();
    return initialValue;
  });

  return (
    <div>
      <p>{count}</p>
      <button onClick={() => setCount(count +
1)}>Increment</button>
    </div>
  );
};

export default Example;
```

In this example, the useState hook takes a function that returns the initial state value. The function is executed only once when the component is mounted, and its return value is used as the initial state. This can be useful if you need to perform some expensive computation to calculate the initial state value, as it allows you to avoid doing that computation on every render.

It's worth noting that this pattern is not commonly used in React, and most of the time, you can simply pass the initial state value directly as an argument to useState.

22- How to manipulate dom with refs?

Ans- Here's an example of how to use refs to manipulate a DOM element:

```
import React, { useRef } from 'react';

const Example = () => {
  const inputRef = useRef(null);

  const handleClick = () => {
```

```
    // Focus the input element
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={handleClick}>Focus the input</button>
    </div>
  );
};

export default Example;
```

In this example, we use the useRef hook to create a ref that we can attach to an input element. We attach the ref to the input using the ref prop, and then we use the ref in the handleClick function to focus the input when the button is clicked.

here's another example of using refs in React:

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

const Example = () => {
  const messageRef = useRef(null);

  useEffect(() => {
    // Scroll to the bottom of the message container
    const container = messageRef.current;
    container.scrollTop = container.scrollHeight;
  }, []);

  return (
    <div ref={messageRef} style={{ overflow: 'auto', height:
'200px' }}>
      <p>Message 1</p>
      <p>Message 2</p>
      <p>Message 3</p>
      {/* Add more messages... */}
    </div>
  );
};
```

```
export default Example;
```

In this example, we use a ref to access a message container and scroll to the bottom of it whenever the component is mounted. The ref is created using the useRef hook and attached to the container using the ref prop. In the useEffect hook, we use the ref to scroll to the bottom of the container whenever the component is mounted, we can also add a variable in the useEffect dependency array that contains the messages getting rendered, this will allow the screen to scroll to the bottom with every new added message. This can be useful in chat applications, for example, to make sure that the user always sees the latest messages.

23- what are keys in react and how do they improve performance?

Ans- In React, keys are used to uniquely identify elements in an array of elements. When you render a list of elements in React, it's important to assign a key to each element so that React can keep track of the elements and update them efficiently.

Here's an example of using keys in React:

```
import React from 'react';

const Example = () => {
  const items = [
    { id: 1, text: 'Item 1' },
    { id: 2, text: 'Item 2' },
    { id: 3, text: 'Item 3' },
  ];

  return (
    <ul>
      {items.map(item => (
        <li key={item.id}>{item.text}</li>
      ))}
    </ul>
  );
};

export default Example;
```

In this example, we're rendering a list of items, and each item has a unique id property. We use the id property as the key for each item, which helps React keep track of the items and update them efficiently. If we didn't provide a key, React would

have to re-render the entire list of items whenever the list changes, which can be slow and inefficient. By using keys, React can keep track of which items have changed and update them accordingly, thereby improving performance.

It's worth noting that keys should be unique among sibling elements, but they don't need to be globally unique. A good practice is to use the unique identifier of each item, such as an id property, as the key. If the items don't have unique identifiers, you can use the index of the item in the array as a key, but this is not recommended, as it can lead to bugs and inefficiencies when the order of the items changes.

24- Is it recommended to use index as a key in react? Explain.

Ans- When React renders a list of elements, it uses the keys to keep track of the elements and to update the DOM efficiently.

Keys should be unique among the siblings, meaning that they should be different for each item in the list, even if the items are the same.

If two items have the same key, React will not be able to determine which item has changed, leading to possible bugs and slow performance.

Using the index of an item as its key can lead to inefficiencies and slow performance for several reasons:

1. **Re-ordering elements:** If the order of elements in the list changes, React will have to re-render all of the elements, as the keys will have changed. This can lead to slow performance and inefficiencies, especially with large lists.
2. **Unstable keys:** If you add or remove an item from the list, the index of the remaining items will change, and React will have to re-render all of the elements. This can lead to slow performance and inefficiencies, especially with large lists.
3. **Performance optimization:** React uses the keys to optimize updates and avoid re-rendering unchanged elements. If the keys are not unique or stable, React will not be able to optimize updates efficiently.

Therefore, it's generally not recommended to use the index of an item as its key in React. Instead, it's better to use a unique identifier for each item, such as an id property, as its key. This way, React can keep track of the elements and update them efficiently, leading to better performance and fewer bugs.

25- What is the difference between controlled and uncontrolled components?

Ans- In React, the term "controlled component" refers to a form element whose value is controlled by the state of a functional component, and "uncontrolled component" refers to a form element that maintains its own state internally.

Here's an example of a controlled component using a functional component and the useState hook:

```
const ControlledInput = () => {
  const [inputValue, setInputValue] = useState("");

  const handleInputChange = (event) => {
    setInputValue(event.target.value);
  };

  return (
    <input
      type="text"
      value={inputValue}
      onChange={handleInputChange}
    />
  );
};
```

In this example, the useState hook is used to manage the state of the input, with the initial value being an empty string. The handleInputChange function updates the state whenever the user types in the input, and the value prop of the input is set to inputValue.

Here's an example of an uncontrolled component using a functional component:

```
const UncontrolledInput = () => {
  const handleSubmit = (event) => {
    event.preventDefault();
    console.log(event.target.elements.input.value);
  };

  return (
    <form onSubmit={handleSubmit}>
```

```
      <input type="text" defaultValue="initial value" name="input" />
      <button type="submit">Submit</button>
    </form>
  );
};
```

In this example, the input's value is not controlled by the component's state, but by the DOM. The component can still access the value of the input by using event.target.elements.input.value in the handleSubmit function, but it has no direct control over the value.

In general, controlled components are preferred in React because they make it easier to keep track of the state of a form and to ensure that the input values are always valid. However, uncontrolled components can be useful in certain situations, such as when working with legacy code.

26- What does one-way data binding mean in react?

Ans- One-way data binding in React refers to the concept that data can flow from the component's state or props to the component's render method or return value, but changes to the rendered or returned values are not automatically propagated back to the state. This means that the component's state acts as the "single source of truth" for the component, and any changes to the component's UI must be explicitly reflected in the state.

Here's an example of one-way data binding using a functional component:

```
const Example = () => {
  const [text, setText] = useState("Initial value");

  const handleInputChange = (event) => {
    setText(event.target.value);
  };

  return (
    <input type="text" value={text} onChange={handleInputChange}
/>
  );
};
```

In this example, the useState hook is used to manage the component's state. The

initial value of the state is "Initial value". The handleInputChange function updates the state with the new value of the input whenever the user types in the input. This updates the value of the input, ensuring that the component's state and the UI are always in sync.

One-way data binding helps to ensure that the component's state is always accurate, and that any changes to the UI are deliberate and intentional. By using one-way data binding, developers can be confident that the component's state accurately reflects the UI, and that any changes to the state will result in corresponding changes to the UI.

27- What are some limitations of react?

Ans- Some limitations are:

1- Virtual DOM limitations: Although the virtual DOM is a key feature of React, it can also cause performance issues when dealing with large and complex data sets.

Here are some scenarios when the virtual DOM can cause performance issues:

1. Large number of components: When there are a large number of components in a React application, the virtual DOM must manage and update all of these components, which can take a significant amount of time and resources.
2. Frequent state updates: When the state of a component changes frequently, it can result in a high number of virtual DOM updates, which can cause performance issues, especially on low-end devices or browsers.

2- Unpredictable performance: React's performance depends on many factors, such as the size of the application, the number of components, and the complexity of the state. This can make it difficult to predict how the application will perform in different situations.

3- Integration with other libraries: React is just one part of a larger ecosystem of libraries and tools, and integrating it with other libraries can be challenging, especially for developers who are new to the React ecosystem.

28- What is a higher order component in react?

Ans- A higher-order component (HOC) in React is a function that takes a component as an argument and returns a new component with additional functionality. HOCs are

used to reuse code, add additional functionality to a component, or abstract common behavior into a separate component.

Here's an example of a simple HOC in React using a functional component:

```
const withAuth = (Component) => {
  return (props) => {
    if (!props.isAuthenticated) {
      return <Redirect to="/login" />;
    }

    return <Component {...props} />;
  };
};

const MyComponent = (props) => {
  return <div>Secret Content</div>;
};

const MyComponentWithAuth = withAuth(MyComponent);
```

In this example, the withAuth HOC takes a component as an argument and returns a new component that wraps the original component. The new component checks if the user is authenticated by checking the isAuthenticated prop. If the user is not authenticated, the new component returns a Redirect component to the login page. If the user is authenticated, the new component passes on all of its props to the original component.

The HOC can be used in the following way:

```
const App = () => {
  return (
    <div>
      <MyComponentWithAuth isAuthenticated={true} />
    </div>
  );
};
```

Here, the App component passes an isAuthenticated prop of true to the MyComponentWithAuth component, which allows the component to render its secret content. If the isAuthenticated prop was set to false, the HOC would redirect the user to the login page.

29- when to use class over functional component in react or vice versa?

Ans- In React, the choice between using a class component or a functional component is largely a matter of personal preference and the requirements of the particular component. Here are a few guidelines to help you choose between the two:

1. **State:** If a component needs to maintain internal state or lifecycle methods, a class component is required.
2. **Performance:** Functional components are generally faster than class components due to their simpler implementation and lack of overhead associated with class-based components. However, in cases where a component needs to manage state or perform complex calculations, the difference in performance is negligible.
3. **Readability:** Functional components are often easier to read and understand, especially for simple components that only render based on props. Class components can become more complex, especially as they grow in size and include more lifecycle methods.
4. **Reusability:** If a component needs to be shared or reused in multiple places, it is often easier to implement it as a functional component.

Ultimately, the choice between using a class component or a functional component will depend on the specific requirements of each component and your own personal coding style and preferences. In many cases, it is possible to write equivalent functionality using either a class component or a functional component, so it is important to choose the approach that is most clear and maintainable for your codebase.

30- What are stateless and stateful components?

Ans- In React, components can be either class components or functional components.
Stateless components, also known as functional components, are components that receive data (props) and render it to the UI, but do not maintain their own state. They are simple and easy to write and are often used for presentational purposes.

Here's an example of a stateless component in React:

```
function DisplayName(props) {
  return <h1>Hello, {props.name}</h1>;
```

```
}
```

On the other hand, stateful components, are components that maintain their own state and can update it as the user interacts with the UI. They are used for complex and dynamic UI.
Here's an example of a stateful component in React:

```
function App(props) {
const [name,setName] = useState('')

return (
  <div>
    <button onClick={()=> setName('john')}>change name</button>
    <DisplayName name={name}/>
  </div>
)}
```

31- What is the cleanup function in useEffect? Why is it used? How does it help?

Ans- The cleanup function in useEffect is a function that gets executed before the component using useEffect is unmounted or before the effect is re-run. It's used to undo any changes made in the effect and to clean up any resources that were allocated during the effect.

Here's an example of a useEffect with a cleanup function:

```
useEffect(() => {
  const intervalId = setInterval(() => {
    // ...
  }, 1000);

  // Cleanup function
  return () => clearInterval(intervalId);
}, []);
```

In the example above, the cleanup function is returning a function that stops the interval that was started in the effect. This is important because if the component using the effect gets unmounted before the interval stops, it would result in a memory leak. The cleanup function ensures that the interval is stopped and that any

resources used in the effect are cleaned up.

The cleanup function is optional and only necessary if the effect creates or modifies some external resources that need to be cleaned up. If there's no cleanup needed, the useEffect can be written without it.

32- What is a memory leak in react? Explain with example.

Ans- A memory leak in React occurs when a functional component that is no longer being used continues to take up memory resources. This can lead to slow performance or crashes if the memory usage continues to increase over time.

Here's an example of a memory leak in a functional component:

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

const ExampleComponent = () => {
  useEffect(() => {
    const intervalId = setInterval(() => {
      console.log('Hello');
    }, 1000);

    // No cleanup function to clear the interval
  });

  return <div>Hello World</div>;
}

export default ExampleComponent;
```

In this example, the useEffect hook is used to set an interval that logs "Hello" to the console every second. However, there is no cleanup function provided to clear the interval when the component is unmounted. As a result, the interval continues to run even after the component is no longer being used, creating a memory leak.

To fix the memory leak, you can provide a cleanup function to clear the interval when the component is unmounted:

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

const ExampleComponent = () => {
  useEffect(() => {
    const intervalId = setInterval(() => {
      console.log('Hello');
```

```
    }, 1000);

    return () => clearInterval(intervalId);
  });

  return <div>Hello World</div>;
}

export default ExampleComponent;
```

In this example, the cleanup function clearInterval(intervalId) is returned from the useEffect hook to clear the interval when the component is unmounted, avoiding the memory leak.

33- What is the cleanup function in useEffect? Why is it used? How does it help?

Ans- The cleanup function in useEffect is an optional function that is executed when a component that uses useEffect unmounts or when the component re-renders and the useEffect hook needs to be re-run. It's used to undo any side effects that the component's useEffect hook performed during its execution.

The purpose of the cleanup function is to avoid any unintended consequences when a component that uses useEffect unmounts or re-renders. It helps to clean up any resources, subscriptions, timers, or event listeners that the component created during its execution so that they do not continue to run and consume memory or cause other issues.

Here's a code example to demonstrate how the cleanup function works in useEffect:

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

const Example = () => {
  const [width, setWidth] = useState(window.innerWidth);

  useEffect(() => {
    const handleResize = () => setWidth(window.innerWidth);
    window.addEventListener("resize", handleResize);

    return () => {
      // Cleanup function
      window.removeEventListener("resize", handleResize);
    };
  }, []);
```

```
  return (
    <div>
      <p>Window width: {width}</p>
    </div>
  );
};

export default Example;
```

In this example, useEffect adds an event listener to the window's resize event, which updates the component's width state whenever the window is resized. The cleanup function removes the event listener when the component unmounts or the hook is re-run, avoiding any memory leaks or unintended behavior.

34- What is lazy loading (code splitting) in react?

Lazy loading (also known as code splitting) in React is a technique for loading parts of a web application only when they are needed, instead of loading everything at once. This can significantly improve the performance of a web application, especially for large or complex applications, by reducing the amount of JavaScript code that needs to be downloaded and parsed by the browser.

Lazy loading in React is achieved by using the React.lazy component, which allows a component to be loaded dynamically when it is first rendered, instead of being included in the main bundle of JavaScript code that is loaded when the application starts.

Here's an example of how to use React.lazy:

```
import React, { lazy, Suspense } from "react";

const OtherComponent = lazy(() => import("./OtherComponent"));

const Example = () => (
  <div>
    <Suspense fallback={<div>Loading...</div>}>
      <OtherComponent />
    </Suspense>
  </div>
);

export default Example;
```

In this example, the OtherComponent is loaded lazily when it is first rendered, and it

is wrapped in a Suspense component, which provides a fallback UI to display while the component is being loaded. The import statement inside the lazy function is used to dynamically load the component's code, which will only be downloaded and parsed by the browser when it is needed.

35- Talk about some commonly used custom hooks in react.

Ans-

1. **useWindowSize hook:** useWindowSize is a custom hook that can be used to get the current size of the window in a React application. The hook can be used to detect changes in the window size, allowing you to update your components accordingly.

Here is an example of a useWindowSize hook:

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

function useWindowSize() {
  const [windowSize, setWindowSize] = useState({
    width: window.innerWidth,
    height: window.innerHeight,
  });

  useEffect(() => {
    function handleResize() {
      setWindowSize({
        width: window.innerWidth,
        height: window.innerHeight,
      });
    }

    window.addEventListener('resize', handleResize);
    return () => window.removeEventListener('resize', handleResize);
  }, []);

  return windowSize;
}
```

In the above example, the useWindowSize hook returns an object with the current width and height of the window. The hook uses the useState and useEffect hooks to detect changes in the window size and update the state accordingly. The useEffect hook adds a resize event listener to the window and removes it when the component using the hook is unmounted.

2. **useStorage hook:** useStorage is a custom hook that allows you to easily interact with the browser's local storage in a React application. The hook provides a simple API for reading and writing data to local storage, without having to manually deal with the complexities of the localStorage API.

Here is an example of a useStorage hook:

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

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

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

  return [value, setValue];
}
```

In the above example, the useStorage hook takes a key argument and returns an array with two values: the current value stored in local storage for that key, and a function to update the value. The hook uses the useState and useEffect hooks to synchronize the value stored in local storage with the state. The useEffect hook updates the value in local storage whenever the state changes.

You can use the useStorage hook in your components like this:

```
function MyComponent() {
  const [data, setData] = useStorage('my-data');

  // ...
}
```

3. **useDebounce hook:** useDebounce is a custom hook that allows you to debounce a function in a React component. Debouncing is a technique for controlling the rate at which a function gets executed, so that it doesn't run too often. This is useful when you have an event that generates multiple calls to a function, and you want to avoid executing that function multiple times in a short interval.

Here is an example of a useDebounce hook:

Note- Debouncing can be used in a search input to avoid sending a new search

request for every change in the input value. Instead, the search request is delayed for a certain amount of time, so that the input can stabilize and the user can finish typing before the search is executed.

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

function useDebounce(value, delay) {
  const [debouncedValue, setDebouncedValue] = useState(value);

  useEffect(() => {
    const handler = setTimeout(() => {
      setDebouncedValue(value);
    }, delay);

    return () => {
      clearTimeout(handler);
    };
  }, [value, delay]);

  return debouncedValue;
}
```

In the above example, the useDebounce hook takes two arguments: value and delay. value is the value to debounce, and delay is the number of milliseconds to wait before updating debouncedValue. The hook uses the useState and useEffect hooks to debounce the value. The useEffect hook sets a timeout that updates debouncedValue with the latest value after the specified delay interval.

You can use the useDebounce hook in your components like this:

```
function MyComponent() {
  const [inputValue, setInputValue] = useState('');
  const debouncedInputValue = useDebounce(inputValue, 500);

  // ...
}
```

In this example, debouncedInputValue will update only once every 500 milliseconds, even if inputValue changes multiple times in a shorter interval.

36- What are the different ways a component can be re-rendered in react?

Ans- In React, a component can be re-rendered in the following ways:

1. **State changes:** When the component's state changes, the component will re-render. This can happen when the setState method is called, or when a state update is triggered by a custom hook.
2. **Props changes:** When the component's props change, the component will re-render. This can happen when the parent component passes new props to the child component.
3. **Context changes:** When the context value changes, components that consume the context will re-render.