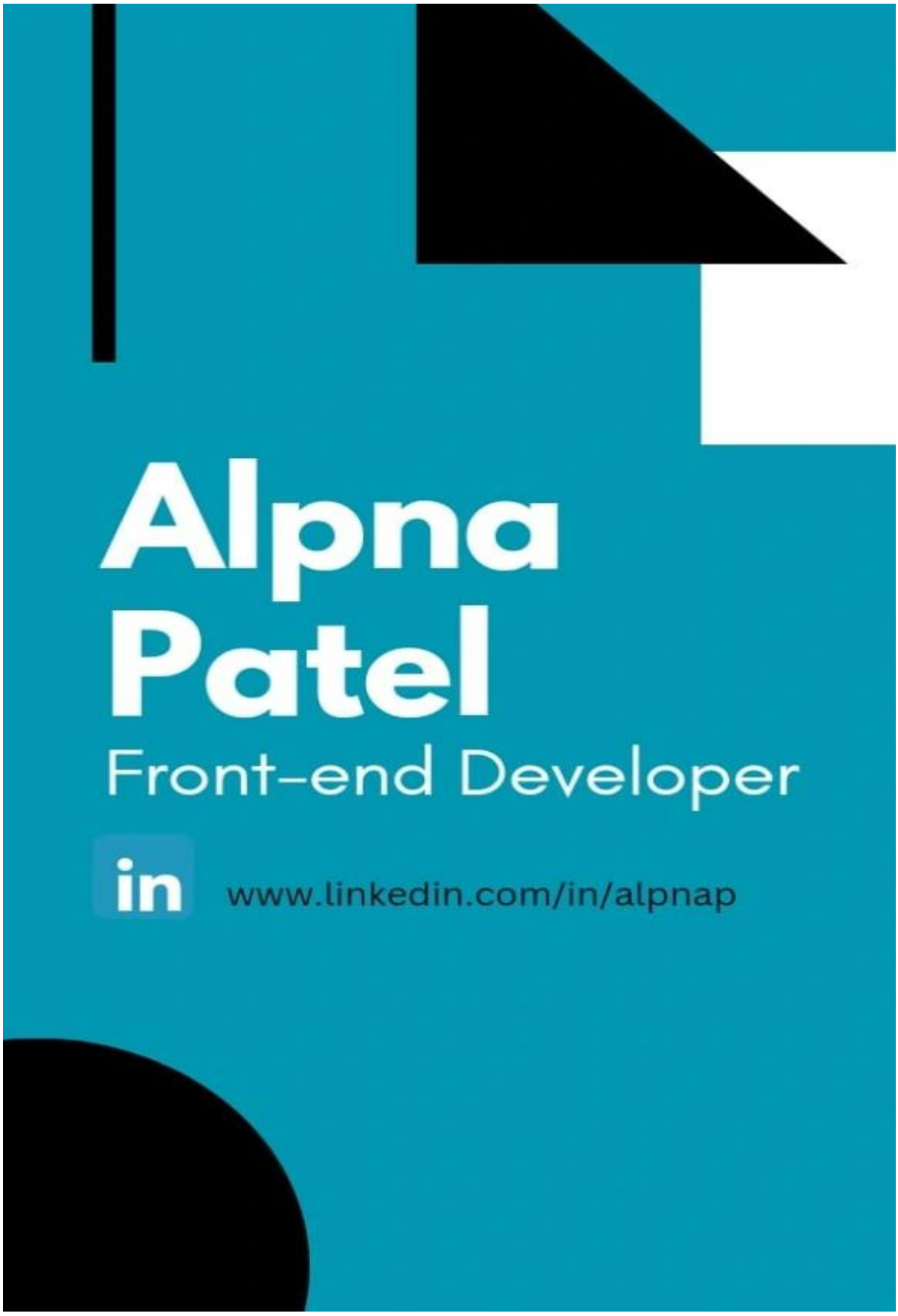

Alpna
Patel
Front-end Developer
in
www.linkedin.com/in/alpnap

REACT JS INTERVIEW QUESTIONS

With answers to crack the technical interview

All the best !

Mastering React: Essential Interview Questions and Answers for Developers

1. What is React?

Answer: React is a JavaScript library developed by Facebook for building user interfaces, particularly for single-page applications. It allows developers to create dynamic and responsive web applications through a component-based architecture.

Key features of React include:

1. **Component-Based Architecture**: React encourages the development of reusable UI components. Each component manages its own state and can be composed to create complex user interfaces.
2. **JSX Syntax**: React uses JSX, which allows developers to write HTML-like code within JavaScript. This makes the code more intuitive and easier to understand.
3. **Virtual DOM**: React maintains a virtual representation of the DOM, which optimizes rendering performance. When a component's state changes, React updates the virtual DOM first, compares it with the previous version, and then efficiently updates only the parts of the real DOM that changed.
4. **Unidirectional Data Flow**: Data in React flows in one direction, from parent to child components. This makes it easier to understand how data changes affect the UI, leading to more predictable application behavior.
5. **Hooks**: Introduced in React 16.8, hooks allow developers to use state and other React features in functional components, promoting cleaner and more reusable code.

React is widely adopted in the industry due to its flexibility, performance, and strong community support, making it a preferred choice for modern web development.

2. What is JSX?

Answer: JSX, or JavaScript XML, is a syntax extension for JavaScript that is primarily used with React. It allows developers to write HTML-like code within JavaScript, making it easier to create and visualize the structure of user interfaces.

Key points about JSX include:

1. HTML-Like Syntax: JSX resembles HTML, which makes it intuitive for developers to understand and design UI components. This syntax helps in quickly conceptualizing how the components will look.

2. JavaScript Expressions: You can embed JavaScript expressions directly within JSX by using curly braces `{}`. This allows for dynamic content generation, enabling you to incorporate variables, functions, and other expressions into your markup.

3. Transpilation: Browsers do not understand JSX natively, so it needs to be transpiled into regular JavaScript using tools like Babel. During this process, JSX code is converted into `React.createElement()` calls, which generate the corresponding React elements.

4. Component Nesting: JSX allows for the easy nesting of components, creating a clear hierarchy. This feature makes it simple to compose complex UIs from smaller, reusable components.

5. Attributes and Styles: In JSX, attributes use camelCase syntax instead of HTML's kebab-case. For example, `className` is used in place of `class`, and inline styles are provided as objects.

In summary, JSX enhances the development process in React by providing a more readable and expressive way to define UI components, bridging the gap between JavaScript and HTML.

3. What is DOM?

Answer: The Document Object Model (DOM) is a programming interface that represents the structure of an HTML or XML document as a tree of objects. It allows programming languages, particularly JavaScript, to interact with and manipulate the content, structure, and style of web pages dynamically.

Key aspects of the DOM include:

1. Tree Structure: The DOM organizes a document into a hierarchical tree structure, where each node represents elements, attributes, and text. This structure facilitates easy navigation and manipulation of the document.

2. Dynamic Interaction: With the DOM, JavaScript can dynamically change the content and style of web pages in response to user actions, such as clicks and form submissions, without requiring a page reload. This capability enhances user experience by creating interactive web applications.

3. API for Manipulation: The DOM provides a set of methods and properties that enable developers to:

- Select elements (e.g., `document.getElementById`, `document.querySelector`).
- Modify element attributes and content.

- Create and delete elements.

- Handle events (e.g., mouse clicks, keyboard events).

4. Browser Compatibility: While different browsers may have slight variations in their implementation, the DOM adheres to standards set by the World Wide Web Consortium (W3C), ensuring a consistent experience across different platforms.

5. Relation to Virtual DOM: In libraries like React, a virtual DOM is used as an in-memory representation of the actual DOM. This approach optimizes performance by minimizing direct manipulations of the real DOM, allowing for efficient updates and rendering.

4. What is Virtual Dom?

Answer: The Virtual DOM is an in-memory representation of the actual Document Object Model (DOM) used in web applications, particularly in libraries like React. It serves as a lightweight copy of the real DOM, enabling more efficient updates and rendering of the user interface.

Key aspects of the Virtual DOM include:

1. **Efficiency**: Instead of directly manipulating the real DOM, which can be slow and resource-intensive, the Virtual DOM allows for updates to be made in memory first. This reduces the number of costly direct DOM manipulations.
2. **Diffing Algorithm**: When a component's state changes, the Virtual DOM is updated first. React then compares the new Virtual DOM with the previous version using a process called "diffing." This algorithm identifies what has changed, allowing React to determine the minimal set of updates required for the real DOM.
3. **Batch Updates**: By batching multiple updates together, React can efficiently apply changes to the real DOM in a single operation. This reduces rendering time and improves overall performance.
4. **Improved User Experience**: The use of the Virtual DOM results in smoother and faster updates, leading to a more responsive user interface. Users experience less lag and more fluid interactions, even in complex applications.
5. **Abstraction from the Browser**: The Virtual DOM abstracts away the complexities of the actual DOM, allowing developers to focus on building components without worrying about the underlying performance implications.

In summary, the Virtual DOM is a core concept in React that optimizes rendering performance and enhances the user experience by minimizing direct interactions

with the real DOM. It allows developers to build dynamic and responsive applications more efficiently.

5. What are fragments in React?

Answer: Fragments in React are a feature that allows developers to group multiple elements without introducing additional nodes to the DOM. This is useful for returning multiple elements from a component without needing a wrapping element, such as a .

Syntax: Fragments can be created using React.Fragment or the shorthand syntax with empty tags (<> </>).

```
// Using React.Fragment
return (
  <React.Fragment>
    <Child1 />
    <Child2 />
  </React.Fragment>
);

// Using shorthand
return (
  <>
    <Child1 />
    <Child2 />
  </>
);
```

6. What are props in React?

Answer: Props, short for "properties," are a core feature in React that allow components to receive data and configuration from their parent components. They enable components to be dynamic and reusable by passing data and functions as arguments.

Key points about Props include:

1. **Unidirectional Data Flow**: Props facilitate a one-way data flow, meaning that data is passed down from parent components to child components. This makes it easier to manage and understand how changes in data affect the UI.

2. **Read-Only**: Props are immutable, which means that a child component cannot modify the props it receives. This immutability ensures predictable behavior and helps maintain the integrity of data throughout the application.
3. **Passing Data**: Props can include a variety of data types, such as:
    - Strings, numbers, and booleans
    - Arrays and objects
    - Functions (for event handling)
4. **Default Props**: You can define default values for props using the defaultProps property. This provides fallback values in case certain props are not provided by the parent.
5. **Prop Types**: To enforce type checking, you can use the PropTypes library. This helps ensure that components receive the correct types of data, enhancing code quality and maintainability.
6. **Children Prop**: React also provides a special children prop, allowing you to pass nested elements directly within the component tags. This enables more flexible component composition.

For example:

```
function Parent() {
  const name = "Alice";
  return <Child name={name} />;
}

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

7. What is state in React JS?

Answer: State in React is a built-in object that allows components to maintain and manage their own dynamic data. Unlike props, which are passed from parent to child components, state is local to the component and can be modified by the component itself.

Key points about State include:

1. **Local to the Component**: Each component can have its own state, allowing it to manage data that is specific to that component. This means different components can have different states and behave independently.
2. **Mutable**: State is mutable, meaning it can be updated within the component. When the state changes, React automatically re-renders the component to reflect the new state, ensuring the UI is always in sync with the underlying data.
3. **Using State**: In functional components, state is typically managed using the useState hook:

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

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

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

4. **Lifecycle Management**: State changes can be tracked through component lifecycle methods in class components (e.g., componentDidMount, componentDidUpdate). This allows developers to perform actions based on state changes.
5. **Performance**: Since state updates trigger re-renders, it's important to manage state carefully to optimize performance, especially in larger applications. React's reconciliation process helps ensure that only the necessary components are updated.
6. **Complex State Management**: For more complex state management across components, developers may use libraries like Redux or the Context API to manage global state effectively.

8. What are hooks in React?

Answer: Hooks are a feature in React that allow developers to use state and other React features in functional components, making it easier to manage component logic and promote code reuse. Introduced in React 16.8, hooks enable functional

components to handle state and lifecycle methods, which were previously only available in class components.

Key points about Hooks include:

1. **State Management**: The useState hook allows you to add state to functional components. It returns an array containing the current state value and a function to update that state:

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

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

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

2. **Side Effects**: The useEffect hook manages side effects in functional components, such as fetching data or subscribing to events. It runs after the render and can be configured to trigger based on specific dependencies:

```
useEffect(() => {
  const fetchData = async () => {
    // Fetch data here
  };
  fetchData();
}, []); // Runs once after the first render
```

3. **Custom Hooks**: Developers can create custom hooks to encapsulate and share logic across multiple components. This promotes code reuse and keeps components clean:

```
function useFetch(url) {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch(url)
      .then(response => response.json())
      .then(data => setData(data));
  }, [url]);

  return data;
}
```

4. **Rules of Hooks**: There are two primary rules to follow:
    - Only call hooks at the top level of a functional component or from other custom hooks.
    - Only call hooks from React function components, not from regular functions.
5. **Other Built-in Hooks**: React provides several other built-in hooks, such as:
    - useContext: To access context values.
    - useReducer: For managing complex state logic.
    - useRef: To persist values across renders without causing re-renders.

9. What are components in React?

Answer: Components are the core building blocks of a React application, enabling the creation of reusable and modular pieces of the user interface. They encapsulate both the structure (HTML) and behavior (JavaScript) of a part of the UI, allowing for a clear separation of concerns and improved maintainability.

Key points about components include:

1. **Reusability**: Components can be reused across the application, reducing redundancy and promoting consistency. Once a component is defined, it can be instantiated multiple times with different props.
2. **Composition**: Components can be nested within other components, allowing developers to build complex UIs from simple building blocks. This

composition enables a hierarchical structure that mirrors the layout of the application.

3. **State and Props**:
    - **State**: Components can maintain their own state using hooks (like useState in functional components) or this.state in class components. This state can be dynamic and respond to user interactions.
    - **Props**: Components receive data and configuration through props, which are passed down from parent components. Props are read-only and facilitate communication between components.
4. **Lifecycle Management**: Class components have lifecycle methods (e.g., componentDidMount, componentDidUpdate) that allow developers to run code at specific points in a component's lifecycle. Functional components can achieve similar functionality using the useEffect hook.
5. **Types of Components**:
    - **Functional Components**: Defined as JavaScript functions, these are simpler and often preferred for their readability and ease of use. They can utilize hooks for state and lifecycle management.

```
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}
```

    - **Class Components**: Defined as ES6 classes that extend React.Component, these are more complex and provide access to lifecycle methods.

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

6. **Higher-Order Components**: These are functions that take a component and return a new component, allowing for enhanced behavior or additional functionality.

10. What are the differences between State and Props in React?

Answer: State and props are fundamental concepts in React, each serving distinct roles within components. Here are the key differences:

1. **Definition**:
    - **State**: State is a built-in object that holds data specific to a component. It can change over time, usually in response to user actions or events, and is used to manage the component's internal data.
    - **Props**: Props (short for properties) are used to pass data and event handlers from a parent component to a child component. They are read-only and allow components to communicate with each other.
2. **Mutability**:
    - **State**: State is mutable. It can be updated using the setState method (in class components) or the updater function returned by the useState hook (in functional components). When the state changes, the component re-renders to reflect the new state.
    - **Props**: Props are immutable within the component that receives them. A child component cannot modify its props; it can only read and use them.
3. **Usage**:
    - **State**: Used for managing dynamic data that affects rendering within the component, such as user inputs or fetched data.
    - **Props**: Used to pass data and functions to child components, enabling communication and configuration of child behavior.
4. **Component Ownership**:
    - **State**: Owned and managed by the component itself. Each component can have its own state that it controls independently.
    - **Props**: Owned by the parent component. The parent defines the props and passes them to child components.

11. What are props drilling and How we can avoid props drilling?

Answer: Props drilling refers to the process of passing data through multiple layers of components in a React application, from a parent component to deeply nested child components. While this method allows for data sharing, it can lead to complexity and challenges in managing the code.

Key points about props drilling include:

1. **Hierarchical Data Passing**: In props drilling, data is passed from a parent component down through each level of the component tree. Each

intermediate component must accept the props and then pass them down, even if those components don't use the data directly.

2. **Complexity**: As the component tree deepens, props drilling can become cumbersome. This leads to:
    - Increased boilerplate code, as each intermediate component must forward props.
    - Difficulty in tracking where data is coming from and how it is used.
    - Reduced clarity and maintainability of the codebase.
3. **Example**:

```
function Grandparent() {
  const data = "Hello from Grandparent";
  return <Parent data={data} />;
}

function Parent({ data }) {
  return <Child data={data} />;
}

function Child({ data }) {
  return <h1>{data}</h1>;
}
```

4. **Solutions**: To avoid props drilling, developers can use several strategies:
    - **Context API**: This allows sharing data across components without passing props through every level. You can create a context and consume it in any child component.
    - **State Management Libraries**: Libraries like Redux or MobX can manage state globally, making it accessible to any component without the need for props drilling.
5. **Performance Considerations**: While props drilling itself may not directly impact performance, it can lead to unnecessary rerenders if state is managed at a high level. This can be mitigated using techniques like React.memo or optimizing the component structure.

In summary, props drilling is a method of passing data through multiple component layers in React. While effective, it can introduce complexity and maintenance challenges, which can be alleviated using the Context API or state management libraries.

12. What are Pure components in React?

Answer: Pure components in React are components that optimize rendering by implementing a shallow comparison of their props and state. If neither the props nor the state have changed, the component will not re-render, which can significantly improve performance by avoiding unnecessary updates.

Key points about Pure Components include:

1. **Shallow Comparison**: Pure components perform a shallow comparison of their props and state. This means that if the references for the props or state have not changed, the component will skip re-rendering. This is particularly beneficial for components that receive the same props frequently.
2. **Implementation**:
    - **Class Components**: To create a pure component, you can extend React.PureComponent instead of React.Component:

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

    - **Functional Components**: For functional components, you can use React.memo, which wraps the component and provides the same shallow comparison functionality:

```
const MyFunctionalComponent = React.memo(function({ value }) {
  return <div>{value}</div>;
});
```

3. **Performance Optimization**: By preventing unnecessary re-renders, pure components help keep the application responsive and efficient, especially in cases where the component tree is complex or components are rendered frequently.

4. **Limitations**: It's important to note that pure components only perform shallow comparisons. This means that if props or state are complex data structures (like arrays or objects), you need to ensure immutability. If a nested object is modified without changing its reference, the component may still re-render unnecessarily.
5. **Use Cases**: Pure components are particularly useful in applications with complex UI structures where certain components may often receive unchanged data. They can help reduce the rendering workload and enhance overall performance.

13. What are Error boundaries?

Answer: Error boundaries are a feature in React designed to catch JavaScript errors that occur in the component tree, preventing the entire application from crashing. They enable developers to handle errors gracefully and provide a fallback UI when an error occurs in child components.

14. What are Higher order components in React?

Answer: Higher-Order Components (HOCs) are an advanced pattern in React that allows you to enhance or modify the behaviour of a component. An HOC is a function that takes a component as an argument and returns a new component with additional features or functionality.

Key points about Higher-Order Components include:

1. **Definition**: A Higher-Order Component is essentially a function that wraps a component, adding new props, state, or behavior. It follows the pattern:

```
const withExtraProps = (WrappedComponent) => {
  return (props) => {
    const extraProps = { extraData: "Some extra data" };
    return <WrappedComponent {...props} {...extraProps} />;
  };
};
```

2. **Purpose**: HOCs are used for:
    - **Code Reuse**: Encapsulating shared logic to be reused across multiple components.
    - **Conditional Rendering**: Adding logic to determine when to render a component or modify its data.
    - **Props Manipulation**: Adding, modifying, or filtering props before passing them to the wrapped component.
3. **Example**: Here's a simple example of using an HOC to add extra props:

```jsx
const Greeting = ({ name, extraData }) => {
  return <h1>Hello, {name}! {extraData}</h1>;
};

const EnhancedGreeting = withExtraProps(Greeting);

const App = () => {
  return <EnhancedGreeting name="Alice" />;
};
```

4. **Naming Convention**: HOCs are typically named with a with prefix (e.g., withLogging, withAuth) to indicate that they enhance the wrapped component.
5. **Limitations**:
    - HOCs do not automatically pass refs down to wrapped components.
    - Overusing HOCs can lead to "wrapper hell," complicating the component tree and making it harder to debug.
6. **Common Use Cases**: HOCs are often used for:
    - Data fetching (e.g., withDataFetching).
    - Authentication checks (e.g., withAuth).
    - Logging or analytics (e.g., withLogging).

   In summary, Higher-Order Components are a powerful tool in React for enhancing component functionality and promoting code reuse. Understanding HOCs can help you write more modular and maintainable React applications.

15. What are the differences between controlled and uncontrolled components?

Answer: In React, controlled and uncontrolled components are two approaches to handling form inputs. Here are the key differences:

1. **Definition**:
    - **Controlled Components**: In controlled components, the form data is handled by the React component's state. The value of the input is controlled by React, meaning that the input's value is set by the state, and any changes to the input are communicated back to the state through event handlers.
    - **Uncontrolled Components**: In uncontrolled components, the form data is managed by the DOM itself. React does not control the input's value, and you can access the input's value using refs when needed.

2. **State Management**:
    - **Controlled Components**: The component's state drives the input's value, which means you need to define an onChange handler to update the state whenever the input changes.

```
const ControlledInput = () => {
  const [value, setValue] = useState('');

  const handleChange = (e) => {
    setValue(e.target.value);
  };

  return <input type="text" value={value} onChange={handleChange} />;
};
```

    - **Uncontrolled Components**: The input maintains its own state. You can access the value of the input using a ref when you need to submit or read it.

```
const UncontrolledInput = () => {
  const inputRef = useRef(null);

  const handleSubmit = () => {
    alert(`Input value: ${inputRef.current.value}`);
  };

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

3. **Form Submission**:
    - **Controlled Components**: Since the input value is held in the component's state, you can easily access and validate the data when submitting the form.
    - **Uncontrolled Components**: To access the input value during submission, you typically use refs, which can be less straightforward compared to controlled components.

4. **Advantages**:
    - **Controlled Components**:
        - More predictable as the state is centralized.
        - Easier to implement validation and conditional rendering.
    - **Uncontrolled Components**:
        - Simpler to set up for simple use cases.
        - Less boilerplate code since you don't need to manage state for every input.
5. **Disadvantages**:
    - **Controlled Components**:
        - Can become verbose with multiple inputs.
        - May lead to performance issues if not handled properly, especially in large forms.
    - **Uncontrolled Components**:
        - Less predictable since the state is not centralized in React.
        - More challenging to validate inputs or perform operations based on their values.

In summary, controlled components use React state to manage form inputs, providing better predictability and easier validation. Uncontrolled components allow the DOM to handle input state, offering simplicity for basic forms. The choice between them depends on the specific use case and complexity of the form.

16. What is useCallback?

Answer: useCallback is a built-in React hook that allows you to memoize a callback function, optimizing performance by preventing unnecessary re-creations of the function on every render. It is particularly useful in scenarios where you pass functions as props to child components, which may cause unnecessary renders if the function is re-created each time the parent component renders.

Key Points about useCallback:

1. **Syntax**:

```
const memoizedCallback = useCallback(() => {
  // Your callback logic
}, [dependencies]);
```

2. **Parameters**:
    - **Function**: The first argument is the callback function you want to memoize.
    - **Dependencies Array**: The second argument is an array of dependencies. The memoized function will only be re-created if one of the dependencies has changed since the last render.
3. **Use Cases**:
    - **Performance Optimization**: It helps prevent unnecessary re-renders of child components that rely on reference equality to avoid updates.
    - **Event Handlers**: When passing event handlers to deeply nested components or third-party libraries that depend on stable function references.
4. **Example**: Here's a simple example demonstrating useCallback:

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

const Counter = ({ increment }) => {
  return <button onClick={increment}>Increment</button>;
};

const App = () => {
  const [count, setCount] = useState(0);

  // Memoizing the increment function
  const increment = useCallback(() => {
    setCount((prevCount) => prevCount + 1);
  }, []); // No dependencies means it won't change

  return (
    <div>
      <h1>Count: {count}</h1>
      <Counter increment={increment} />
    </div>
  );
};

export default App;
```

5. **Performance Considerations**:
    - Using useCallback can prevent unnecessary renders, but it also introduces some overhead for maintaining the memoized function. Therefore, it

should be used judiciously—only in cases where you are experiencing performance issues due to excessive re-renders.

17. What are the differences between useMemo and useCallback?

Answer: useMemo and useCallback are both React hooks used for performance optimization, but they serve different purposes and have distinct use cases. Here's a breakdown of their differences:

Feature	`useMemo`	`useCallback`
Purpose	Memoizes a computed value.	Memoizes a callback function.
Return Value	Returns the memoized value.	Returns the memoized function.
Syntax	`const memoizedValue = useMemo(() => computeValue(), [dependencies]);`	`const memoizedCallback = useCallback(() => { /* function logic */ }, [dependencies]);`
Use Case	Use when you want to optimize expensive calculations or derived state.	Use when you want to optimize functions, especially when passing them as props to child components.
Dependencies	The dependencies array determines when the value should be recalculated.	The dependencies array determines when the function should be recreated.
Performance Impact	Helps avoid recalculating values unnecessarily.	Helps avoid re-creating functions unnecessarily, which can prevent re-renders of child components.

18. What is Lazy loading in React?

Answer: Lazy loading is a design pattern in React that helps improve application performance by deferring the loading of components until they are needed. This technique reduces the initial loading time of the application, leading to a better user experience.

Key Aspects of Lazy Loading in React:

1. **Code Splitting**: Lazy loading allows you to split your application into smaller chunks (bundles). This means only the necessary code for the initial render is loaded, while other components are loaded on demand, which can significantly speed up the initial load time.
2. **Using React.lazy**: React provides a built-in function called React.lazy() to facilitate lazy loading. This function takes a dynamic import statement that returns a Promise. For example:

```
const LazyComponent = React.lazy(() => import('./LazyComponent'));
```

3. **Wrapping with Suspense**: To handle the loading state while the lazy component is being fetched, you must wrap the lazy-loaded component in a Suspense component. You can provide a fallback UI (like a loading spinner) to enhance user experience:

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

const LazyComponent = React.lazy(() => import('./LazyComponent'));

const App = () => {
  return (
    <div>
      <h1>My App</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent />
      </Suspense>
    </div>
  );
};
```

4. **Benefits**:
    - **Improved Performance**: By loading only necessary components, lazy loading decreases the initial load time, making the application faster and more responsive.
    - **Enhanced User Experience**: Users can start interacting with parts of the application that are already loaded while other components are being fetched.
5. **Common Use Cases**:
    - Loading large components or libraries that are not needed immediately.
    - Dynamically importing routes in single-page applications (SPAs) to optimize navigation.
    - Any component that is resource-heavy and can be loaded on demand.
6. **Limitations**:
    - There can be a slight delay when a lazy-loaded component is first accessed, as it needs to be fetched.

- Proper management of loading states is essential to ensure a smooth user experience.

In summary, lazy loading in React is a powerful technique to optimize performance by deferring the loading of components until they are required. By using React.lazy() in conjunction with Suspense, developers can enhance load times and provide a better overall experience for users.

19. What is suspense in React ?

Answer: Suspense is a feature in React that allows you to handle the loading states of components more elegantly, particularly when dealing with asynchronous operations like code splitting or data fetching. It provides a way to specify a loading fallback UI while a component is being loaded, improving the user experience.

Benefits:

- Simplifies the process of handling loading states.
- Enhances user experience by allowing you to show meaningful fallback UIs while waiting for content to load.
- Makes it easier to manage asynchronous operations in a declarative way.

20. What are custom hooks?

Answer: Custom hooks in React are a way to encapsulate and reuse stateful logic across different components. They are essentially JavaScript functions that leverage React's built-in hooks (like useState, useEffect, etc.) and allow you to create reusable logic that can be shared between components.

Key Points:

1. **Definition**: A custom hook is a function that begins with the prefix use and can call other hooks. This naming convention is important because it helps React identify the function as a hook.

```
function useCustomHook() {
  // Hook logic goes here
}
```

2. **Purpose**: The main goal of custom hooks is to promote code reuse and separation of concerns. They help keep your components cleaner and more focused on rendering UI rather than managing complex logic.
3. **Creating a Custom Hook**: To create a custom hook, define a function that uses other hooks. For example, here's a simple custom hook for fetching data:

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

function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch(url);
        if (!response.ok) throw new Error('Network response was not ok');
        const result = await response.json();
        setData(result);
      } catch (err) {
        setError(err);
      } finally {
        setLoading(false);
      }
    };

    fetchData();
  }, [url]);

  return { data, loading, error };
}
```

4. **Using a Custom Hook**: You can use your custom hook in a functional component just like any other hook:

```jsx
const App = () => {
  const { data, loading, error } = useFetch('https://api.example.com/data');

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error: {error.message}</div>;

  return <div>{JSON.stringify(data)}</div>;
};
```

5. **Benefits**:
   - **Code Reusability**: Custom hooks enable you to share logic without duplicating code.
   - **Separation of Concerns**: They help keep components focused on rendering by abstracting away complex logic.
   - **Encapsulation**: Custom hooks encapsulate functionality, making it easier to manage and test.
6. **Best Practices**:

- Always prefix custom hooks with use to follow React conventions.
- Ensure each hook focuses on a specific functionality to maximize reusability.
- Properly manage side effects and cleanups within your custom hooks.

21. What is useReducer hook ?

Answer: The useReducer hook is a built-in React hook that is used for managing state in functional components. It is an alternative to useState and is particularly useful for handling complex state logic or when the next state depends on the previous state.

Key Aspects of useReducer:

1. **Purpose**: useReducer is primarily used for state management in scenarios where:
    - The state is complex and consists of multiple sub-values.
    - State updates depend on previous state values.
    - You want to implement state transitions that are more predictable, similar to how Redux works.
2. **Syntax**:

```
const [state, dispatch] = useReducer(reducer, initialState);
```

    - reducer: A function that determines how the state changes in response to an action. It takes the current state and an action as arguments and returns the new state.
    - initialState: The initial state value.
3. **Reducer Function**: The reducer function is where the state transitions occur. It typically follows this pattern:

```
function reducer(state, action) {
  switch (action.type) {
    case 'ACTION_TYPE':
      return { ...state, /* updated values */ };
    default:
      return state;
  }
}
```

4. **Dispatching Actions**: To update the state, you call the dispatch function with an action object. The action object usually has a type property and can also carry

```
dispatch({ type: 'ACTION_TYPE', payload: {/* data */} });
```

5. **Benefits**:
    - **Predictable State Management**: State transitions are centralized in the reducer function, making it easier to track changes.
    - **Separation of Logic**: Encourages separation of state management logic from UI logic.
    - **Complex State Handling**: Suitable for managing complex state that involves multiple sub-values.
6. **When to Use**: Use useReducer when:
    - You have complex state logic that involves multiple actions or sub-states.
    - The state relies on previous state values.
    - You want a more structured approach to state management.

22. Which lifecycle hooks in class component are replaced with useEffect in functional components ?

Answer: In React, the useEffect hook in functional components can replicate the behavior of several lifecycle methods from class components. Here's how they correspond:

1. **componentDidMount**:
    - This method is called once, immediately after the component is mounted.
    - **Equivalent in useEffect**:

```
useEffect(() => {
  // Code here runs after the component mounts
}, []); // Empty dependency array ensures it runs only once
```

2. **componentDidUpdate**:
    - This method is invoked after the component updates due to changes in state or props.

- **Equivalent in useEffect**:

```
useEffect(() => {
  // Code here runs after every render/update
}); // Without a dependency array, it runs after every render

// For specific updates, you can include dependencies:
useEffect(() => {
  // Code runs only when specific dependencies change
}, [dependency]); // Runs when 'dependency' changes
```

3. **componentWillUnmount**:
    - This method is called just before the component is unmounted and removed from the DOM.
    - **Equivalent in useEffect**:

```
useEffect(() => {
  return () => {
    // Cleanup code runs just before the component unmounts
  };
}, []); // Empty dependency array ensures this runs on unmount
```

In summary, the useEffect hook effectively combines the functionalities of componentDidMount, componentDidUpdate, and componentWillUnmount from class components. By using dependencies, you can precisely control when your effects run, making useEffect a powerful tool for managing side effects in functional components.

23. What is context in React?

Answer: Context in React is a powerful feature that allows you to share values such as data or functions between components without having to pass props through every level of the component tree. This is particularly useful for managing global data that multiple components need to access, such as themes, user authentication, or language settings.

Key Points:

1. **Purpose**:

- Context helps avoid "prop drilling," which is the tedious process of passing props down through many layers of components. It allows for easier data sharing across the application.

2. **Creating Context**:
   - You create a context using React.createContext(), which returns a Context object that contains two components: Provider and Consumer.

```
const MyContext = React.createContext();
```

3. **Provider**:
   - The Provider component wraps parts of the component tree that need access to the context. It accepts a value prop that holds the data you want to share.

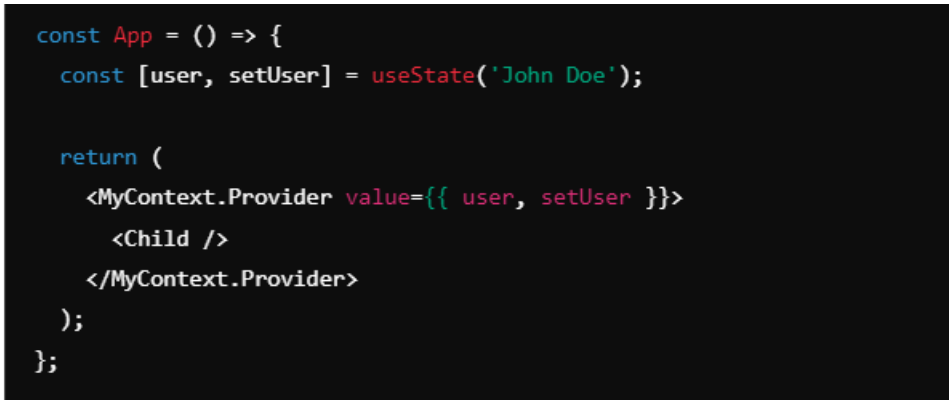
```
const App = () => {
  const value = { user: 'John Doe' };

  return (
    <MyContext.Provider value={value}>
      <Child />
    </MyContext.Provider>
  );
};
```

4. **Consumer**:
   - Components that need to access the context can use the Consumer component or the useContext hook (for functional components) to retrieve the context value.

```
const Child = () => {
  return (
    <MyContext.Consumer>
      {({ user }) => <div>Hello, {user}!</div>}
    </MyContext.Consumer>
  );
};
```

5. **Updating Context**:

- You can make the context value dynamic by using state management within the Provider, allowing components to update the context value.



```
const App = () => {
  const [user, setUser] = useState('John Doe');

  return (
    <MyContext.Provider value={{ user, setUser }}>
      <Child />
    </MyContext.Provider>
  );
};
```

6. **Benefits**:
    - Reduces the need for prop drilling.
    - Centralizes state management for global data.
    - Simplifies component composition.
7. **Considerations**:
    - Context should be used judiciously to avoid performance issues, as any component subscribing to the context will re-render when the context value changes.

In summary, Context in React is an essential tool for sharing data among components without extensive prop drilling. It promotes cleaner, more maintainable code, making it easier to manage global application state. However, it's important to use it thoughtfully to maintain performance and reduce unnecessary re-renders.

24. What are some lifecycle methods in React?

Answer: In React class components, lifecycle methods are special methods that allow you to run code at specific points in a component's lifecycle. Here are some key lifecycle methods:

1. **componentDidMount**:
    - This method is called immediately after a component is mounted (inserted into the tree). It's commonly used for data fetching, setting

up subscriptions, or initializing any processes that require the component to be present in the DOM.

```
componentDidMount() {
  // Fetch data or set up subscriptions here
}
```

2. **componentDidUpdate**:
    - This method is called immediately after updating occurs. It receives the previous props and state as arguments, allowing you to respond to changes in data and trigger side effects based on those changes.

```
componentDidUpdate(prevProps, prevState) {
  // Respond to prop or state changes here
}
```

3. **componentWillUnmount**:
    - This method is called immediately before a component is unmounted and destroyed. It's used for cleanup tasks, such as removing event listeners or canceling any ongoing network requests.

```
componentWillUnmount() {
  // Clean up resources here
}
```

4. **shouldComponentUpdate**:
    - This method is called before rendering when new props or state are received. It allows you to optimize performance by controlling whether a component should update or not based on specific conditions.

```
shouldComponentUpdate(nextProps, nextState) {
  return nextProps.value !== this.props.value; // Example condition
}
```

5. **getDerivedStateFromProps**:
    - This static method is invoked right before rendering and can return an object to update state based on props. It's useful for synchronizing state with changes in props.

```
static getDerivedStateFromProps(nextProps, prevState) {
  // Return new state based on props here
}
```

6. **getSnapshotBeforeUpdate**:
    - This method is invoked right before the rendered output is committed to the DOM. It allows you to capture information (a snapshot) from the DOM, which can be used in componentDidUpdate.

```
getSnapshotBeforeUpdate(prevProps, prevState) {
  // Capture some information from the DOM
}
```

In summary, lifecycle methods in React class components provide a way to manage and respond to different stages of a component's life, enabling tasks such as data fetching, performance optimization, and cleanup. Understanding these methods is essential for effectively managing side effects and component behavior. In functional components, similar behavior can be achieved using the useEffect hook.

25. How can you optimize performance in a React app?

Answer: Optimizing performance in a React application is crucial for ensuring a smooth user experience. Here are several strategies to achieve this:

1. **Use React.memo**:
    - Wrap functional components with React.memo to prevent unnecessary re-renders. It does a shallow comparison of props, only re-rendering if the props change.

```
const MyComponent = React.memo((props) => {
  // Component code
});
```

2. **Implement Pure Components**:
    - For class components, use PureComponent instead of Component. This automatically implements a shallow comparison of props and state, optimizing rendering.

```
class MyComponent extends React.PureComponent {
  // Component code
}
```

3. **Use the useCallback and useMemo Hooks**:
    - Utilize useCallback to memoize callback functions, preventing them from being recreated on every render.
    - Use useMemo to memoize expensive calculations, avoiding recalculation on every render unless dependencies change.

```
const memoizedCallback = useCallback(() => {
  // Callback code
}, [dependencies]);

const memoizedValue = useMemo(() => {
  // Expensive calculation
}, [dependencies]);
```

4. **Code Splitting**:
    - Use dynamic imports and React.lazy for code splitting, which allows loading parts of the application only when needed, reducing the initial load time.

```
const LazyComponent = React.lazy(() => import('./LazyComponent'));
```

5. **Optimize Rendering**:
    - Avoid rendering unnecessary components. Use conditional rendering to display components only when needed.
    - Utilize keys in lists to help React identify which items have changed, are added, or are removed, which can improve performance.
6. **Avoid Inline Functions in Render**:
    - Define functions outside of the render method to avoid creating new function instances on every render, which can lead to unnecessary re-renders.
7. **Throttling and Debouncing**:
    - Implement throttling or debouncing for high-frequency events like scrolling or typing to limit the number of updates triggered.

```
const handleScroll = _.throttle(() => {
  // Scroll handling logic
}, 200);
```

8. **Optimize State Management**:
    - Lift state up to avoid unnecessary re-renders of child components.
    - Use local state for component-specific data to minimize the impact on the global state.
9. **Use Production Builds**:
    - Always use the production build of your React app for deployment. Development builds include extra warnings and checks that can slow down performance.
10. **Performance Monitoring**:
    - Use tools like React Profiler to analyze performance bottlenecks and identify which components are rendering excessively.

Summary

In summary, optimizing performance in a React application involves a combination of techniques such as memoization, code splitting, careful state management, and performance monitoring. By applying these strategies, you can significantly enhance the user experience by reducing load times and minimizing unnecessary re-renders.