Here's a **comprehensive list of basic ReactJS interview questions** designed to evaluate foundational knowledge in ReactJS development:

General ReactJS Concepts

1. What is React? Why is it used?

React is an open-source JavaScript library developed by Facebook for building user interfaces, particularly single-page applications.

It focuses on creating reusable UI components, making it easier to manage the complexity of modern web applications.

Key Features of React

- **Component-Based Architecture**:
- **Virtual DOM**
- **Unidirectional Data Flow**
- **JSX Syntax**

Uses of React:

1. **Efficiency with Virtual DOM**:
   ○ React minimizes the number of direct manipulations to the real DOM, which can be slow. The Virtual DOM ensures updates are optimized.
2. **Reusable Components**:
   ○ Components can be reused across different parts of the application, speeding up development and ensuring consistency.
3. **Developer Productivity**:
   ○ With tools like React DevTools and a supportive ecosystem, debugging and developing with React is streamlined.
4. **Strong Community and Ecosystem**:
   ○ React has a vast community, which means plenty of libraries, resources, and third-party tools are available.
5. **Flexibility**:

○ React can be used with other libraries or frameworks, such as Redux for state management or Next.js for server-side rendering.

6. **Rich User Interfaces**:

○ React makes it easier to build complex, dynamic, and interactive user interfaces efficiently.

---

2. What is the difference between React and other JavaScript frameworks like Angular or Vue?

Features	ReactJS	Angular
Type	Library for building UIs.	Full-fledged framework.
Design	Focuses on UI; other functionalities need third-party libraries.	Provides a complete solution (routing, state management, etc.).
Architecture	Component-based, with unidirectional data flow.	Component-based, but includes bidirectional data binding.
Tooling Complexity	Requires multiple tools (e.g., Redux, React Router).	All-in-one with built-in solutions.
State management	Third-party (e.g., Redux, MobX, Zustand).	Built-in (RxJS, NgRx).
Routing	React Router (third-party).	Built-in with Angular Router.
Rendering	Virtual DOM ensures efficient updates.	Real DOM, but optimized with Zones and change detection.
Data binding	One-way data binding by default.	Two-way data binding with `ngModel`.
When to use	Best for SPAs, dynamic UIs, and when flexibility is needed.	Ideal for large, enterprise-level applications.

3. What are the advantages and limitations of using React?

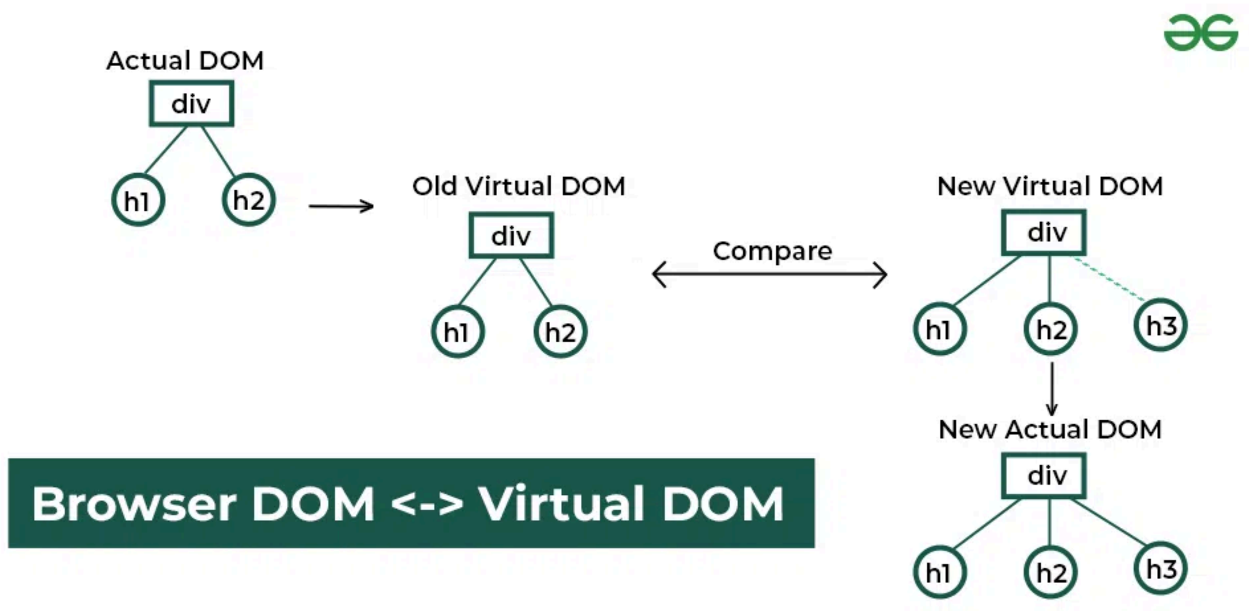
Limitations:-

- React often requires additional libraries for state management, routing, or form handling, which can lead to a higher setup complexity.
- Many third-party libraries in the React ecosystem may have inadequate documentation or inconsistent quality.
- React relies heavily on external libraries for functionalities like state management (e.g., Redux, MobX) or server-side rendering (e.g., Next.js).
- React applications that rely solely on client-side rendering can have SEO issues unless tools like Next.js are used for server-side rendering.

4. Explain the concept of Virtual DOM in React.

In React, the Virtual DOM (VDOM) serves as this blueprint for your user interface.

- When your React component renders for the first time, React creates a virtual DOM, a lightweight JavaScript object representation of the actual DOM.
- This virtual DOM is a copy of the real DOM, but it exists entirely in memory.
- Whenever a state or prop changes in your component, React re-renders the component, creating a new virtual DOM.
- React's efficient diffing algorithm compares the old virtual DOM with the new one, identifying the minimal set of changes required to update the real DOM.
- This process is highly optimized to minimize the number of DOM manipulations



Real-World Example

Without Virtual DOM:

Imagine you want to update a list of 100 items in the Real DOM:

- Without optimization, the browser may re-render the entire list, even if only one item changes.

With Virtual DOM:

- React detects that only one item's content has changed, updates it in the Virtual DOM, and applies just that change to the Real DOM.

Importance of using virtual DOM:

Performance Optimization:

a. Direct DOM manipulation can be slow, especially for large-scale applications.
b. The virtual DOM allows React to batch updates and only update the necessary parts of the real DOM, significantly improving performance.

In essence, the virtual DOM acts as a layer of abstraction between your React components and the actual DOM, enabling efficient and performant UI updates.

5. What is JSX, and why is it used in React?

JSX (JavaScript XML) is a **syntax extension for JavaScript** that allows developers to write HTML-like code directly within JavaScript.

JSX is not a requirement for React, but it makes code more readable and expressive.

Importance of JSX

JSX allows you to write HTML-like code within JavaScript. This makes it easier to visualize the UI components directly in the logic where they're created.

Since JSX is embedded in JavaScript, you can seamlessly integrate JavaScript expressions within your UI code.

JSX is transpiled by tools like **Babel** into standard JavaScript (`React.createElement()` calls), which React uses to build the Virtual DOM.

JSX is not valid JavaScript—it must be **transpiled** (converted) into standard JavaScript before execution.

For example, the above JSX code:

```
<div>

   <h1>Hello, {name}!</h1>

 </div>
```

is transpiled to:

```
    React.createElement('div', null,

    React.createElement('h1', null, `Hello, ${name}!`)

    );
```

Features of JSX

1. **Embedding Expressions**:

JavaScript expressions can be embedded in curly braces `{}`:

```
<h1>{2 + 2}</h1> // Renders: <h1>4</h1>
```

o

2. **Attributes**:

JSX allows setting attributes just like HTML:

```
<img src="image.jpg" alt="Description" />
```

3. **Conditional Rendering**:

Use expressions for conditional rendering:

```
<p>{isLoggedIn ? 'Welcome back!' : 'Please log in.'}</p>
```

○

4. **Styling**:

Inline styles can be applied using objects:

```
<div style={{ color: 'red', fontSize: '20px' }}>Styled Text</div>
```

5. **Nested Components**:

JSX allows nesting components easily:

```
<Header />

<Footer />
```

Advantages of JSX

- Writing UI in JSX feels more natural and closely resembles HTML.
- JSX is tailored to React, providing seamless integration with its component-based architecture.
- Since JSX is embedded in JavaScript, you can utilize its full power for loops, conditionals, or calculations.

Limitation:

- JSX must be transpiled using tools like Babel, adding a build step to the workflow.
- Deeply nested components or complex UIs can make JSX verbose and harder to read.

6. Can browsers read JSX directly? If not, how is it converted?

Example: Transpilation of JSX

JSX Code:

```
const App = () ⇒ {
   return (

     <div>

       <h1>Hello, World!</h1>

     </div>

   );

 };
```

Transpiled JavaScript Code:

```
const App = () => {

   return React.createElement(

     'div',

     null,

     React.createElement('h1', null, 'Hello, World!')

   );

 };
```

React.createElement:

a. This function creates a JavaScript object representing the Virtual DOM node.
b. The object includes the element type (`'div'` or `'h1'`), any attributes (`null` in this case), and child nodes (`Hello, World!`).

Tools for Transpiling JSX

1. **Babel:**
   ○ Babel is the most commonly used tool for transpiling JSX into JavaScript.
   ○ It uses a preset like `@babel/preset-react` to understand JSX syntax.

Example Babel Configuration:

json

Copy code

```json
{

  "presets": ["@babel/preset-react"]

}
```

2. **Build Tools**:

○ Tools like **Webpack**, **Vite**, or **Parcel** integrate Babel to handle JSX during the build process.

3. **CDN-Based Transpilation**:

○ For simple projects, tools like **esbuild** or online CDNs (e.g., Babel Standalone) can transpile JSX directly in the browser (not recommended for production).

Real-Time Transpilation for Development

During development, JSX is typically transpiled in real-time:

Development Mode:

c. Tools like React Scripts (from Create React App) handle this automatically using Babel and Webpack.

Production Mode:

d. The transpiled code is minified and optimized before deployment.

7. What are React components? What are their types?

In React, a **component** is a reusable, self-contained piece of the user interface (UI).

Each component represents a part of the UI, such as a button, a form, or an entire page, and can be combined to build complex UIs.

Components are responsible for:

Rendering the UI: Defining what should appear on the screen.

Managing State: Handling data that changes over time.

Handling Logic: Managing user interactions, fetching data, or processing inputs.

Types of React Components

React components are broadly categorized into two types:

- **Functional Components**
- **Class Components**

Functional components are JavaScript functions that accept `props` (inputs) and return React elements describing what should appear on the UI.

Introduced Features:

- Initially, functional components were stateless.
- With the introduction of **React Hooks**, functional components can now manage state and lifecycle methods.

```
import React from 'react';

const Greeting = ({ name }) => {

 return <h1>Hello, {name}!</h1>;

};

export default Greeting;
```

Key Features:

- Simpler to write and understand.
- Use **React Hooks** (useState, useEffect, etc.) to handle state and lifecycle methods.

- Generally preferred in modern React development.

Class components are ES6 classes that extend `React.Component`. They must include a `render()` method, which returns the React elements to display.

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

class Greeting extends Component {

 render() {

   return <h1>Hello, {this.props.name}!</h1>;

 }

}

export default Greeting;
```

Key Features:

- Include built-in lifecycle methods such as componentDidMount, componentDidUpdate, and componentWillUnmount.
- Manage state using this.state and update it using this.setState.

feature	Functional	Class
Syntax	Functions	ES6 Classes
State Management	Managed using **React Hooks**	Managed using `this.state`
Lifecycle Methods	Handled via **Hooks** (`useEffect`)	Use built-in lifecycle methods
Complexity	Simpler	More boilerplate
Performance	Slightly better (due to no `this` keyword)	Slightly less performant (uses `this`)

Presentational Components:

- Focus only on rendering UI.
- Do not manage state or handle logic.
- Often functional components.

```
const Button = ({ text }) ⇒ <button>{text}</button>;
```

Higher-Order Components (HOCs):

A pattern in React where a function takes a component and returns a new component with additional features.

```
const withLogger = (Component) ⇒ {

   return (props) ⇒ {

     console.log('Props:', props);

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

   };

 };

  const EnhancedComponent = withLogger(MyComponent);
```

Controlled and Uncontrolled Components:

- **Controlled Components:** Manage form inputs using React state.
- **Uncontrolled Components:** Use native DOM methods with refs to manage inputs.

Controlled Example:

```
const Form = () ⇒ {

   const [value, setValue] = React.useState('');
   return <input value={value} onChange={(e) => setValue
(e.target.value)} />;

 };
```

Note:- The reason parameters in React **functional components** are often provided with **curly brackets** {} is to take advantage of **JavaScript destructuring**.

8. What is the difference between functional components and class components?

The key differences between **functional components** and **class components** in React lie in their syntax, features, and usage. Here's a detailed comparison:

---

1. Syntax

Functional Components:

- Written as JavaScript functions.
- Simple and concise.

Example:

```
function Greeting({ name }) {

   return <h1>Hello, {name}!</h1>;

 }
```

Class Components:

- Written as ES6 classes that extend `React.Component`.
- Require a `render()` method to return JSX.

Example:

```
class Greeting extends React.Component {

   render() {

     return <h1>Hello, {this.props.name}!</h1>;

   }

 }
```

---

2. State Management

Functional Components:

- Originally stateless but gained the ability to manage state with **React Hooks** (introduced in React 16.8).
- Use `useState` and `useReducer` hooks to manage state.

Example:

```
function Counter() {

   const [count, setCount] = React.useState(0);

    return (

     <div>

       <p>Count: {count}</p>
```

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

Class Components:

- Manage state using `this.state` and update it with `this.setState`.

Example:

```
class Counter extends React.Component {
   constructor(props) {
     super(props);
     this.state = { count: 0 };
   }
    increment = () ⇒ {
     this.setState({ count: this.state.count + 1 });
   };
    render() {
     return (
       <div>
         <p>Count: {this.state.count}</p>
```

```
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
```

---

3. Lifecycle Methods

Functional Components:

- Handle lifecycle events using **React Hooks** like `useEffect`.

Example (equivalent to `componentDidMount`):

```
function App() {
  React.useEffect(() => {
    console.log('Component Mounted');
     return () => {
      console.log('Component Unmounted'); // Cleanup (like
`componentWillUnmount`)
    };
  }, []); // Empty dependency array ensures this runs only once
   return <h1>Hello, World!</h1>;
}
```

Class Components:

- Use dedicated lifecycle methods like:
  - componentDidMount (called after the component mounts).
  - componentDidUpdate (called after state or props update).
  - componentWillUnmount (called before the component unmounts).

Example:

```
function App() {

   React.useEffect(() ⇒ {

     console.log('Component Mounted');

      return () ⇒ {

       console.log('Component Unmounted'); // Cleanup (like
`componentWillUnmount`)

     };

   }, []); // Empty dependency array ensures this runs only once

    return <h1>Hello, World!</h1>;

 }
```

4. Performance

- **Functional Components**:
  - Slightly more performant because they don't require the overhead of the this keyword or binding methods.
  - Easier to optimize with React.memo for memoization.
- **Class Components**:
  - Slightly less performant due to the use of this and more complex structure.

---

5. Modern React Usage

- **Functional Components**:
  - Preferred in modern React development.
  - Simpler, easier to write, and maintain.
  - Fully capable of managing state and lifecycle using **Hooks**.
- **Class Components**:
  - Used in older codebases.
  - Still supported by React but increasingly less common in new projects.

---

6. Code Readability and Boilerplate

- **Functional Components**:
  - Simpler and more concise.
  - No need for boilerplate like `constructor` or `this`.
- **Class Components**:
  - More verbose.
  - Requires additional boilerplate for constructors, method binding, etc.

Which Should You Use?

- **Use Functional Components**:
  - For new React projects.
  - They are simpler, require less code, and can handle all features (state, lifecycle, etc.) using Hooks.
- **Use Class Components**:

○ When maintaining or updating older React codebases that already use class components.

9. What is the role of React.createElement() in JSX?

In React, `React.createElement()` is a core function responsible for creating the **Virtual DOM elements.**

Browsers cannot interpret JSX directly, so it needs to be converted into standard JavaScript code. This is where `React.createElement()` comes into play.

Signature of `React.createElement()`

React.createElement(type, [props], [...children])

- **type**: The type of the element (e.g., a string for built-in DOM elements like div or h1, or a React component).
- **props**: An object containing the element's properties/attributes (e.g., id, className, onClick).
- **children**: The child elements or content inside the element.

Example

JSX Code:

```
const element = (

 <button className="btn" onClick={() ⇒ alert('Clicked!')}>

   Click Me!

 </button>

);
```

Transpiled Code:

```
const element = React.createElement(
 'button',

 { className: 'btn', onClick: () ⇒ alert('Clicked!') },

 'Click Me!'

);
```

- **Type**: `'button'`
- **Props**: `{ className: 'btn', onClick: () => alert('Clicked!') }`
- **Children**: `'Click Me!'`

Props and State

11. What are props in React? How are they different from state?

In React, **props** (short for "properties") and **state** are core concepts for managing and passing data in components.

Props are read-only data passed from a parent component to a child component.

They are used to configure the child component and make it reusable by providing dynamic values.

To pass data or functions from a parent to a child component.

Immutable (cannot be changed by the receiving component).

Props are typically used to display dynamic data or customize a child component's behavior.

```
function Greeting(props) {

 return <h1>Hello, {props.name}!</h1>;

}
```

```
function App() {
 return <Greeting name="John" />;
}
```

State is a local, mutable data store specific to a component.

It allows components to manage and react to user interactions or other dynamic changes.

To manage and update data within a component over its lifecycle.

State is used to store data that needs to change over time or in response to user actions.

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


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


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

12. How do you pass data from a parent component to a child component?

To pass data from a parent component to a child component in React, you use **props**.

Props allow you to send any data (like strings, numbers, arrays, objects, or functions) as attributes of the child component's JSX tag.

Parent Component:

```
function Parent() {
 const message = "Hello from the Parent!";
 const user = { name: "John", age: 30 };

 return (
   <div>
     <Child greeting={message} userInfo={user} />
   </div>
 );
}
```

Child Component:

```
function Child(props) {
 return (
   <div>
     <p>{props.greeting}</p>
     <p>User: {props.userInfo.name}, Age: {props.userInfo.age}</p>
   </div>
 );
}
```

13. What is state in React? How is it managed?

In React, **state** refers to a component's local, mutable data. It represents dynamic information that can change over time, typically in response to user interactions, API calls, or other events.

When the state of a component changes, React automatically re-renders the component to reflect the updated state.

Key Characteristics of State

- **Local to a Component**: Each component manages its own state independently.
- **Mutable**: State can be updated using React's state management mechanisms.
- **Triggers Re-renders**: Changes to state trigger a re-render of the component and its child components.

```
const [state, setState] = useState(initialValue);
```

Example Form submit:

```
const { useState } = require("react");

export default function App() {

 const [items, setItems] = useState([]);

 const [name, setNames] = useState("");

 const handleSubmit = (e) => {

   e.preventDefault();

   let objItem = { id: items.length + 1, value: name };

   setItems([ ... items, objItem]);
 };
 const handleChange = (e) => {
   let data = e.target.value;
   setNames(data);
 };
 return (
   <div>
     <form onSubmit={handleSubmit}>
    <input type="text" id="name" value={name} onChange={handleChange} />
    <button type="submit">Add Item</button>
     </form>

     <ul>
       {items.map((item) => (
         <li key={item.id}>
           ID: {item.id}, Value: {item.value}
         </li>
       ))}
     </ul>
   </div>
 );
```

```
}
```

14. Explain the difference between controlled and uncontrolled components.

React provides two ways to handle form inputs: **controlled components** and **uncontrolled components**. These approaches determine how the form data is managed and accessed.

---

1. Controlled Components

In controlled components, the **form data is managed by React's state**. The component takes full control of the input element's value using state and updates it through event handlers like `onChange`.

Key Features

- The input value is bound to a state variable.
- React is the "single source of truth."
- Requires an `onChange` handler to update the state.

Example:

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


function ControlledInput() {
 const [name, setName] = useState("");


 const handleChange = (e) ⇒ {
   setName(e.target.value); // Update state as user types
 };
 const handleSubmit = (e) ⇒ {

   e.preventDefault();

   alert(`Name: ${name}`);

 };


 return (

   <form onSubmit={handleSubmit}>

     <input type="text" value={name} onChange={handleChange} />

     <button type="submit">Submit</button>

   </form>

 );

}
```

Pros:

1. Easier to implement validation and logic based on input value.
2. Provides more control over the behavior and appearance of the form.
3. Works well with React's philosophy of component reactivity.

Cons:

1. Requires more boilerplate code to manage state.
2. Can lead to performance issues if there are too many inputs and frequent updates.

2. Uncontrolled Components

In uncontrolled components, the **form data is managed by the DOM itself**. React interacts with the DOM to retrieve the value when needed, usually via a `ref`.

Key Features

- The input value is handled by the DOM.
- React accesses the value when needed, typically on form submission.
- Does not require state or `onChange` handler for the input.

Example:

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


function UncontrolledInput() {

 const nameRef = useRef(); // Reference to the input element


 const handleSubmit = (e) => {

   e.preventDefault();

   alert(`Name: ${nameRef.current.value}`); // Access value via ref

 };


 return (

   <form onSubmit={handleSubmit}>

     <input type="text" ref={nameRef} />

     <button type="submit">Submit</button>
```

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

Pros:

1. Simpler to implement for basic forms where real-time validation or updates aren't needed.
2. Useful for integrating with non-React libraries that directly manipulate the DOM.

Cons:

1. Less control over the input value and behavior.
2. Validation or logic is harder to implement since the value is not directly tied to React's state.
3. Doesn't align well with React's declarative approach.

Key Differences

Aspect	Controlled Component	Uncontrolled Component
Data Handling	Managed by React state.	Managed by the DOM.
Value Access	Accessed via state (`value`).	Accessed via `ref`.

Event Handling	Requires `onChange` to update state.	Not required; value accessed as needed.
Use Case	Complex forms, validation, live updates.	Simple forms, DOM manipulation.
Performance	Can be slower for many inputs.	Generally faster, less React overhead.
Example	`<input value={state} onChange={...} />`	`<input ref={ref} />`

When to Use Which?

Use Controlled Components:

- When you need validation as the user types.
- If you want to perform live updates based on the input.
- For forms with dynamic behavior or state-dependent changes.

Use Uncontrolled Components:

- For simple forms where you only need the value at submission.
- When working with third-party libraries that manipulate the DOM directly.
- For legacy codebases that rely on direct DOM access.

---

15. Can props be modified inside a child component? Why or why not?

No, **props cannot be modified inside a child component** in React. This is because props are **read-only** and are passed down from the parent component to the child component.

Why Props Cannot Be Modified Inside a Child Component

16. **One-Way Data Flow**: React follows a one-way data flow, where data is passed from the parent component to the child component via props. The child component can only receive and read the props but should not modify them. If a child component were able to modify its props, it would break the unidirectional data flow, leading to unpredictable behavior and making the app harder to maintain and debug.
17. **Immutability of Props**: React treats props as immutable, meaning they are not intended to be changed by the receiving component. The parent component has control over the props and can modify them by re-rendering and passing updated props down to the child.
18. **State vs. Props**: If you need to modify data within a child component, you should use **state** instead of props. State is mutable and can be updated within the component. If the child component needs to modify data and communicate changes back to the parent, it can do so by calling a function passed from the parent via props (a "callback function").

19. What happens if you try to update the state directly in React?

If you try to **update the state directly in React**, it can lead to **unexpected behavior** and **bugs**. React state is designed to be **immutable**, meaning it should not be modified directly. Instead, state should always be updated using the **state updater function** (`setState` in class components or the setter function from `useState` in functional components).

Why Directly Modifying State is Problematic

1. **Bypassing React's State Management**: React relies on its internal mechanisms to track state changes and re-render components when necessary. If you modify the state

directly, React won't know that the state has changed, and it won't trigger a re-render. This can result in the UI not reflecting the most recent state.

2. **Potential for Inconsistent State**: React's state management is optimized for performance through techniques like **batching** and **scheduling updates**. If you modify state directly, you bypass this mechanism, which can lead to **inconsistent state** across different renders. React may not properly merge or update the state, causing bugs or incorrect behavior.

3. **Mutating State Directly**: React uses **immutable data structures** for state, meaning that it expects you to treat state as a new value when making updates (instead of modifying the existing value). Direct mutation (like modifying an object or array in place) can lead to **side effects** that are hard to track and debug.

React Lifecycle

18. What are React lifecycle methods?

React lifecycle methods are special functions in React components that allow you to hook into different stages of a component's lifecycle.

These methods are used to initialize components, manage updates, and clean up before the component is removed from the DOM.

The lifecycle of a React component is broadly divided into three phases:

1. Mounting (when the component is created and added to the DOM):

- `constructor(props)`:
  - Invoked before the component is mounted.
  - Used to initialize the state or bind methods.
- `static getDerivedStateFromProps(props, state)`:
  - A static method that runs before `render`.
  - Updates the state based on changes in props.
- `render()`:
  - Responsible for rendering the component's UI.

- This method is **pure** and should not contain side effects.
- `componentDidMount()`:
  - Invoked immediately after the component is added to the DOM.
  - Commonly used for fetching data, starting subscriptions, or integrating third-party libraries.

---

2. Updating (when props or state change):

- `static getDerivedStateFromProps(props, state)`:
  - Also called during the update phase.
- `shouldComponentUpdate(nextProps, nextState)`:
  - Determines if the component should re-render.
  - Used for performance optimizations. By default, it returns `true`.
- `render()`:
  - Re-renders the component with updated state or props.
- `getSnapshotBeforeUpdate(prevProps, prevState)`:
  - Called right before changes are committed to the DOM.
  - Returns a value that is passed to `componentDidUpdate`.
- `componentDidUpdate(prevProps, prevState, snapshot)`:
  - Invoked after the component has updated.
  - Useful for performing DOM operations or additional API calls.

---

3. Unmounting (when the component is removed from the DOM):

- `componentWillUnmount()`:
  - Invoked just before a component is destroyed.
  - Used to clean up resources like event listeners, timers, or subscriptions.

---

4. Error Handling (introduced in React 16):

- `static getDerivedStateFromError(error)`:

  - Used to update the state when an error is caught in a child component.
- `componentDidCatch(error, info)`:
  - Used to log errors or display fallback UI.

Functional Component Equivalent with Hooks

React Hooks provide lifecycle-like features for functional components:

- `useEffect`: Combines the behavior of `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` in one function.
- `useState` and `useReducer`: Used for state management.
- `useMemo` and `useCallback`: Optimize performance by memoizing computations or functions.

Modern React applications increasingly rely on functional components and hooks for managing lifecycle-related tasks.

19. Explain the purpose of componentDidMount() and componentWillUnmount().

Purpose of `componentDidMount()` and `componentWillUnmount()` in Class Components

These lifecycle methods are part of React's **class components** and are used to handle side effects and cleanup tasks.

---

1. `componentDidMount()`

This method is called **once** immediately after the component is added to the DOM (mounted). It is often used for initialization tasks that require the DOM or external data.

Common Use Cases:

- **Fetching Data from an API:** To retrieve data and populate the component's state.

- **Setting Up Subscriptions:** Adding event listeners, like `window.addEventListener` or subscribing to real-time data streams.
- **Updating DOM or Third-Party Integrations:** Initializing libraries, animations, or manipulating the DOM.
- **Triggering Side Effects:** Starting timers, logging analytics, or performing other one-time setup tasks.

Example:

jsx

Copy code

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


class App extends Component {

 state = { data: null };


 componentDidMount() {

   // Fetch data after component mounts

   fetch('https://api.example.com/data')

     .then(response => response.json())

     .then(data => this.setState({ data }));

 }


 render() {

   return <div>Data: {this.state.data || 'Loading...'}</div>;
 }

}


export default App;
```

---

2. `componentWillUnmount()`

This method is called **once** just before the component is removed from the DOM (unmounted). It is used to clean up resources to avoid memory leaks.

Common Use Cases:

- **Removing Subscriptions:** Unsubscribing from real-time data streams or removing event listeners.
- **Clearing Timers or Intervals:** Cancelling `setInterval`, `setTimeout`, or other asynchronous tasks.
- **Cleaning Up Third-Party Integrations:** Destroying external libraries or disconnecting from resources.

Example

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


class Timer extends Component {

 intervalId = null;
```

```
  componentDidMount() {

    // Start a timer when the component mounts

    this.intervalId = setInterval(() ⇒ {

      console.log('Timer tick');

    }, 1000);

  }



  componentWillUnmount() {

    // Clear the timer before the component unmounts

    clearInterval(this.intervalId);

  }



  render() {

    return <div>Timer is running. Check the console for ticks!</div>;

  }

}



export default Timer;
```

20. What is the difference between shouldComponentUpdate() and componentDidUpdate()?

Key Differences

Feature	`shouldComponentUpdate()`	`componentDidUpdate()`
Purpose	Decides if the component should re-render.	Executes side effects after the component updates.
Timing	Before the `render()` method.	After the `render()` method and DOM updates.
Return Value	Must return `true` or `false`.	No return value; used for side effects.
Use Case	Performance optimization (prevent unnecessary renders).	Reacting to updates (e.g., API calls, animations).
Access to Previous Props/State	Not applicable.	`prevProps` and `prevState` are available.

21. Why is the render() method mandatory in class components?

The `render()` method is **mandatory in class components** in React because it is responsible for describing the **structure of the UI** for that component. It is the only required method in a class component and is where the JSX (or React elements) is returned to specify what should be rendered to the DOM.

The `render()` method defines the React component's output. It returns:

22. **JSX**: Describes the component's UI in a declarative manner.
23. **React elements**: A JavaScript representation of the DOM elements.

React's Reconciliation Process

When a class component's state or props change, React:

1. Calls the `render()` method to generate a new "virtual DOM" representation.
2. Compares it with the previous virtual DOM (diffing).
3. Updates only the changed parts of the real DOM (efficient updates).

Without the `render()` method, React cannot generate the virtual DOM or perform reconciliation.

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


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


export default Greeting;
```

Here, the `render()` method:

- Returns JSX that renders the `h1` element.
- React uses this method to determine what to display.

24. How do you handle side effects in class components?

In **class components**, side effects are handled using **lifecycle methods**. Side effects are operations that affect something outside the scope of the function, such as API calls, DOM manipulations, or setting up subscriptions.

Performing Side Effects on Mount

Use the **`componentDidMount()`** lifecycle method for side effects that should occur after the component is added to the DOM.

Example Use Cases:

- Fetching data from an API.
- Setting up event listeners.
- Initializing third-party libraries.

```
class App extends React.Component {

 componentDidMount() {

   // Fetch data after the component is mounted

   fetch('https://api.example.com/data')

     .then((response) => response.json())

     .then((data) => {

       console.log(data);

     });



   // Add an event listener

   window.addEventListener('resize', this.handleResize);

 }



 handleResize = () => {

   console.log('Window resized');

 };



 render() {

   return <div>App Component</div>;
 }

}
```

Transition to Modern React (Hooks)

In **functional components**, the useEffect Hook is used to handle all these side effects, making it more flexible and concise. Here's an equivalent example:

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

function App({ userId }) {

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



 useEffect(() => {

   // Fetch data on mount and when userId changes

   fetch(`https://api.example.com/user/${userId}`)

     .then((response) => response.json())

     .then((data) => setData(data));



   // Cleanup effect

   return () => {

     console.log('Cleanup resources');

   };

 }, [userId]); // Dependency array ensures it runs on userId change



 return <div>{data ? `User: ${data.name}` : 'Loading...'}</div>;
}
```

React Hooks

24. What are React hooks? Why were they introduced?

React Hooks are special functions introduced in **React 16.8** that allow developers to use state and other React features in functional components.

They enable functional components to mimic the behavior of class components, such as managing state, handling side effects, and accessing the lifecycle methods, without writing a class.

Some commonly used hooks include:

25. `useState`: For state management.
26. `useEffect`: For handling side effects like data fetching and subscriptions.
27. `useContext`: For consuming context in a component.
28. `useRef`: For accessing DOM elements or persisting mutable values.
29. `useReducer`: For more complex state management.
30. `useMemo` and `useCallback`: For optimizing performance by memoizing computations or functions.

Hooks were introduced to address several challenges and limitations in React's class components and to enhance the overall developer experience. Here are the key reasons:

1. Simplify State Management in Functional Components

Before hooks, functional components were stateless. To manage state or access lifecycle methods, developers had to use class components, which could be verbose and harder to read for simple logic.

With Hooks: Hooks enable stateful logic and side effects directly in functional components, making them more powerful and reusable.

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

```
function Counter() {

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


 return (

   <div>

     <p>Count: {count}</p>

     <button onClick={() => setCount(count + 1)}>Increment</button>

   </div>

 );

}
```

2. **Eliminate the Complexity of Class Components**

Class components come with their own challenges:

- `this` keyword is often misunderstood and can lead to bugs.
- Managing lifecycle methods (`componentDidMount`, `componentDidUpdate`, etc.) can result in duplicated logic and clutter.

With Hooks: Hooks remove the need for classes entirely, simplifying the learning curve and codebase.

3. **Better Reusability of Logic**

In class components, it was difficult to reuse stateful logic across multiple components. Developers often resorted to:

- **Higher-order components (HOCs):** Can result in nested component trees.

- **Render props**: Can lead to "wrapper hell."

With Hooks: Hooks allow developers to extract reusable logic into **custom hooks**, making it cleaner and easier to share across components.

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


function useFetch(url) {

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


 useEffect(() ⇒ {

   fetch(url)

     .then((response) ⇒ response.json())

     .then((data) ⇒ setData(data));

 }, [url]);


 return data;

}


function App() {

 const data = useFetch('https://api.example.com/data');


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

}
```

Advantages of Hooks

- **No Classes Required**: Simplifies components by eliminating the need for classes.
- **Reusable Logic**: Enables custom hooks for cleaner, reusable code.
- **Improved Readability**: Separates concerns in a way that's easier to understand.
- **Functional Programming Paradigm**: Aligns with modern JavaScript functional patterns.
- **Concurrent Mode Friendly**: Enhances compatibility with React's advanced features.

Limitations of Hooks

- **Learning Curve**: While simpler in many ways, hooks introduce new concepts like dependency arrays in `useEffect` that can be confusing.
- **Overuse of Custom Hooks**: Excessive abstraction with hooks can lead to unclear code.
- **Debugging Challenges**: Tracking hook behavior can sometimes be more complex, especially with nested hooks or dependency array issues.

31. Explain the purpose of the useState hook. Provide an example.

The **`useState`** Hook is a built-in React function that allows functional components to manage state.

It is used to declare a state variable and provides a way to update it.

Purpose of `useState`

32. **Store Data:** It stores and tracks the state of a component, such as user input, toggles, counters, etc.
33. **Trigger Re-renders:** When the state is updated via the setter function, React automatically re-renders the component to reflect the updated state.

34. **Simplify State Management:** Eliminates the need for class components to manage state, making functional components more powerful and concise.

Syntax of `useState`

```
const [state, setState] = useState(initialValue);
```

Advantages of `useState`

35. **Simplicity:** Makes state management in functional components straightforward.
36. **Encapsulation:** Keeps state local to the component where it's declared.
37. **Flexibility:** Can handle various types of state, from simple primitives (numbers, strings) to complex objects and arrays.
38. **Concise Syntax:** Eliminates the need for class-based syntax like `this.state` or `this.setState`.

39. What is the difference between useEffect and useLayoutEffect?

Both `useEffect` and `useLayoutEffect` are React hooks that allow you to perform side effects in functional components. However, they differ in **timing** and **use cases**.

Timing of Execution

- `useEffect`: Runs **after the browser has painted** the screen. This makes it asynchronous with respect to rendering, so it does not block the UI update.
- `useLayoutEffect`: Runs **synchronously after DOM mutations** but **before the browser paints**. This means it blocks the browser from updating the screen until the effect is executed.

---

2. Use Case

- `useEffect`:

○ Ideal for tasks that don't require blocking the DOM rendering, such as:

■ Fetching data from an API.

■ Subscribing to or cleaning up event listeners.

■ Running analytics or logging.

○ It ensures smoother user experiences since it doesn't delay the browser's paint.

- `useLayoutEffect`:

○ Useful when you need to **read and write layout information (synchronously)** after a render, such as:

■ Measuring DOM elements (e.g., dimensions or positions).

■ Performing animations.

■ Making updates that must happen before the browser paints to avoid visual flickering.

---

3. Performance Impact

- `useEffect`:

○ Non-blocking, so it doesn't negatively impact performance or cause layout thrashing.

○ Recommended for most side effects.

- `useLayoutEffect`:

○ Can block rendering if overused, leading to potential performance issues.

○ Use sparingly for critical DOM updates that must happen synchronously.

---

4. Execution in SSR (Server-Side Rendering)

40. `useEffect`:

a. Does not execute during server-side rendering because it's designed to run after rendering is complete.

41. `useLayoutEffect`:

a. Triggers a warning if used in server-side rendering because it depends on the DOM being available, which is not the case in SSR environments.

Using `useEffect` for API Fetching

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


function DataFetcher() {

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


 useEffect(() => {

   fetch('https://api.example.com/data')

     .then((response) => response.json())

     .then((data) => setData(data));

 }, []); // Runs after the component renders


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

}


export default DataFetcher;
```

42. What is the useRef hook, and when would you use it?

The **useRef** hook in React is a built-in hook that returns a **mutable object** called a "ref" that persists across re-renders.

It can hold a **reference** to a DOM element or any mutable value, and it does not trigger a re-render when the value changes.

```
const myRef = useRef(initialValue);
```

`myRef`: A mutable ref object that has a `.current` property.

`initialValue`: The value that the ref will be initialized with (optional).

Key Features of `useRef`

- **Persistent reference**: The ref object persists across renders. This means you can store a value that doesn't change between renders without triggering a re-render.
- **Does not cause re-renders**: Updating the `current` property of a ref object does **not** trigger a re-render of the component.
- **Access DOM elements**: Can be used to reference and manipulate DOM nodes directly, similar to `React.createRef()` in class components.

Accessing DOM Elements

`useRef` is frequently used to store references to DOM elements, so you can interact with them directly without needing to rely on state or props.

Example: Focus an Input Field

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



function FocusInput() {

 const inputRef = useRef(null);



 const focusInput = () ⇒ {

   inputRef.current.focus();  // Directly accessing the DOM element to
focus
```

```
 };


 return (

   <div>

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

     <button onClick={focusInput}>Focus the input</button>

   </div>

 );

}


export default FocusInput;
```

When to Use useRef

- **Accessing or interacting with DOM elements** (e.g., focusing inputs, measuring elements).
- **Storing mutable values** that don't require re-rendering (e.g., previous state, timeout IDs).
- **Integrating with external libraries** or APIs that require direct DOM manipulation.
- **Optimizing performance** by preventing unnecessary re-renders when state updates aren't needed.

43. Explain the role of the useContext hook.

Context is a way to share data (such as themes, authentication status, or language preferences) across the component tree without having to pass props manually at every level.

The **`useContext`** hook in React is used to access data from a **React Context** in a functional component.

The `useContext` hook allows components to "subscribe" to the context and retrieve its current value.

44. What is the useReducer hook, and how is it different from useState?

The `useReducer` hook in React is a powerful tool for managing state, especially when the state logic is complex or involves multiple sub-values

Key Features of `useReducer`

45. **Reducer Pattern**: `useReducer` uses a reducer function, which determines how the state should change in response to an action.
46. **State and Dispatch**: It returns a state variable and a dispatch function. The `dispatch` function is used to send an action to the reducer, which updates the state.
47. **Scalability**: It's well-suited for applications where the state transitions are complex or involve multiple steps or conditions.

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

`reducer`: A function that takes the current state and an action as arguments and returns the new state.

`initialState`: The starting value of the state.

Example of useReducer

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


const initialState = 0;


function reducer(state, action) {

 switch (action.type) {
   case "increment":

     return state + 1;

   case "decrement":

     return state - 1;

   case "reset":

     return initialState;

   default:

     throw new Error("Unknown action type");

 }

}


function Counter() {

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


 return (

   <div>

     <p>Count: {count}</p>

     <button onClick={() => dispatch({ type: "increment"
})}>Increment</button>

     <button onClick={() => dispatch({ type: "decrement"
})}>Decrement</button>

     <button onClick={() => dispatch({ type: "reset" })}>Reset</button>
```

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

48. What are custom hooks? Provide an example.

Custom hooks in React are reusable JavaScript functions that encapsulate logic built using React hooks (`useState`, `useEffect`, `useReducer`, etc.)

They allow you to extract and share stateful logic between components, making your code more modular, readable, and maintainable.

Custom hooks always start with the prefix `use` (e.g., `useCustomHook`). This naming convention is necessary because React relies on it to ensure that the rules of hooks are followed (like only calling hooks at the top level or within a React function component).

Why Use Custom Hooks?

49. **Reusability**: Share logic across multiple components without duplicating code.
50. **Clean Code**: Separate complex logic from the component, keeping it more readable.
51. **Abstraction**: Encapsulate logic into a custom hook, hiding its internal implementation.

Syntax

A custom hook is essentially a function that can call other hooks. Here's a simple template:

```
function useCustomHook() {
  // Hook logic (state, effect, etc.)
  return value; // Return anything you want: state, functions, or objects
}
```

Example of a Custom Hook

1. **Custom Hook for Fetching Data**

useFetch

```
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("Error fetching data");
       const result = await response.json();
       setData(result);
     } catch (err) {
       setError(err.message);
     } finally {
       setLoading(false);
     }
   };

   fetchData();
 }, [url]); // Re-run effect when URL changes

 return { data, loading, error };
}

export default useFetch;
```

Using useFetch in main file

```
import React from "react";
import useFetch from "./useFetch";

function App() {
  const { data, loading, error } =
useFetch("https://jsonplaceholder.typicode.com/posts");

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

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


export default App;
```

2. **Custom Hook for Toggling a Boolean**

useToggle

```
import { useState } from "react";


function useToggle(initialValue = false) {
 const [value, setValue] = useState(initialValue);
 const toggle = () ⇒ setValue((prev) ⇒ !prev);


 return [value, toggle];
}


export default useToggle;
```

Using useToggle

```
import React from "react";
import useToggle from "./useToggle";


function App() {
 const [isVisible, toggleVisibility] = useToggle(false);


 return (
   <div>
     <button onClick={toggleVisibility}>
       {isVisible ? "Hide" : "Show"} Message
     </button>
     {isVisible && <p>Hello, this is a toggleable message!</p>}
   </div>
 );
}


export default App;
```

52. What are the rules of hooks in React?

React Hooks are special functions that allow you to use state and lifecycle features in functional components. To ensure they work as intended, React enforces a set of rules for using hooks. These rules help prevent bugs and maintain predictable behavior in your components.

Rules of Hooks

1. **Only Call Hooks at the Top Level**
   ○ Do not call hooks inside loops, conditions, or nested functions.

Hooks must be called in the same order every time a component renders to ensure proper state association.

```
// Correct

function MyComponent() {

   const [count, setCount] = React.useState(0);

   if (count > 5) {

       // Don't call hooks here

   }

   return <div>{count}</div>;

}


// Incorrect

function MyComponent() {
   if (someCondition) {

       const [count, setCount] = React.useState(0); // Invalid

   }

}
```

Only Call Hooks from React Functions

- Hooks should only be called from:
- ○ React functional components.
- ○ Custom hooks (functions prefixed with `use` that encapsulate hook logic).

```
// Correct

function useCustomHook() {

   const [value, setValue] = React.useState(0);

   return [value, setValue];

}


// Incorrect

function regularFunction() {

   const [value, setValue] = React.useState(0); // Invalid

}
```

Why These Rules Exist

- React relies on the order of Hook calls to associate state and effects with components. Violating these rules can lead to unpredictable behavior and runtime errors.

Using the `eslint-plugin-react-hooks` Linter

- To help enforce these rules, React provides the **eslint-plugin-react-hooks** package. It warns you if you violate the rules of hooks and ensures proper usage in your codebase. Install it by running:

```
npm install eslint-plugin-react-hooks --save-dev
```

Add it to your ESLint configuration:

```
{
   "plugins": ["react-hooks"],
   "rules": {
       "react-hooks/rules-of-hooks": "error", // Checks rules of hooks
       "react-hooks/exhaustive-deps": "warn"  // Checks effect
dependencies
   }
}
```

Event Handling

32. How do you handle events in React? How is it different from plain JavaScript?

n React, handling events is slightly different from handling events in plain JavaScript due to React's use of a virtual DOM and JSX syntax. Here's a detailed breakdown:

Handling Events in React

33. **Event Binding**

a. React uses **camelCase** for event names instead of lowercase as in plain HTML/JavaScript.

```
// React

<button onClick={handleClick}>Click Me</button>



// Plain JavaScript/HTML

<button onclick="handleClick()">Click Me</button>
```

Event Handlers

- In React, you pass a function reference to the event handler rather than a string of JavaScript code.

```
// React

function handleClick() {

   console.log("Button clicked");

}

<button onClick={handleClick}>Click Me</button>



// Plain JavaScript

<button onclick="console.log('Button clicked')">Click Me</button>
```

Using Synthetic Events

- React wraps native DOM events with its **SyntheticEvent** system for cross-browser compatibility and performance optimizations.
  - Example:

```
function handleClick(event) {

   console.log(event); // React's SyntheticEvent

   console.log(event.nativeEvent); // The underlying native DOM event

}

<button onClick={handleClick}>Click Me</button>
```

Prevent Default Behavior

- In React, you call `event.preventDefault()` instead of returning `false` as in plain JavaScript.

```
// React

function handleSubmit(event) {

   event.preventDefault();

   console.log("Form submitted");

}

<form onSubmit={handleSubmit}>

   <button type="submit">Submit</button>

</form>;
```

Passing Arguments

- To pass arguments to an event handler, use an arrow function or the `.bind` method

```
// Using an arrow function

function handleClick(id) {

   console.log("Clicked item with id:", id);

}

<button onClick={() ⇒ handleClick(1)}>Click Me</button>



// Using .bind

<button onClick={handleClick.bind(null, 1)}>Click Me</button>
```

Key Differences from Plain JavaScript

Aspect	React	Plain JavaScript
Event Listener Syntax	Uses camelCase (`onClick`, `onChange`)	Uses lowercase (`onclick`, `onchange`)
Handler Assignment	Pass a function reference	Pass a string of code or a function reference
Event Object	React's `SyntheticEvent`	Native DOM `Event` object
Prevent Default Behavior	Use `event.preventDefault()`	Return `false` or use `event.preventDefault()`
Binding Context (`this`)	Handled by default in function components	Requires explicit binding in some contexts

34. What is synthetic event in React?

In React, a **Synthetic Event** is a wrapper around the native DOM event system that ensures events are consistent and cross-browser compatible. It is part of React's event system, which abstracts away differences in how browsers handle events to provide a uniform API.

```
function App() {

   function handleClick(event) {

       console.log("SyntheticEvent:", event); // SyntheticEvent object

       console.log("NativeEvent:", event.nativeEvent); // Underlying
native DOM event

   }


   return <button onClick={handleClick}>Click Me</button>;

}
```

Event Pooling

React reuses synthetic event objects to improve performance. However, this means you cannot access event properties asynchronously unless you call event.persist().

Without `event.persist()`

```
function App() {

   function handleClick(event) {

       console.log(event.type); // "click"

       setTimeout(() => {
           console.log(event.type); // Error: Properties are nullified

       }, 1000);

   }


   return <button onClick={handleClick}>Click Me</button>;

}
```

With `event.persist()`

```
function App() {

    function handleClick(event) {

        event.persist(); // Prevent pooling

        console.log(event.type); // "click"

        setTimeout(() => {

            console.log(event.type); // "click"

        }, 1000);

    }


    return <button onClick={handleClick}>Click Me</button>;

}
```

35. How do you pass arguments to event handlers in React?

In React, you can pass arguments to event handlers by wrapping the event handler function in another function (commonly using an arrow function) or by using the `.bind()` method. Here's a detailed explanation of both approaches

Using an Arrow Function

An arrow function allows you to call your event handler with arguments while still passing the event to the handler.

```
function App() {

   function handleClick(param) {

      console.log("Button clicked with param:", param);

   }



   return (

      <button onClick={() ⇒ handleClick("Argument 1")}>

         Click Me

      </button>

   );

}
```

How It Works: The arrow function `() => handleClick("Argument 1")` creates a new function that calls `handleClick` with the argument `"Argument 1"`.

Advantages: Simple and concise, especially when passing multiple arguments.

Considerations: A new function is created on every render, which may have performance implications in deeply nested components or frequent re-renders. However, this is usually negligible.

Using `.bind()`

The `.bind()` method can be used to pre-bind arguments to the event handler.

```
function App() {

   function handleClick(param) {

       console.log("Button clicked with param:", param);

   }



   return (

       <button onClick={handleClick.bind(null, "Argument 2")}>

           Click Me

       </button>

   );

}
```

How It Works: The `.bind()` method creates a new function with the specified `param` value bound to the `handleClick` function.

Advantages: Similar to the arrow function but uses explicit binding.

Considerations: Like the arrow function, it creates a new function on each render.

Passing the Event Object Along with Arguments

React automatically provides the event object as the first argument to event handlers. You can pass both custom arguments and the event object by structuring your function accordingly.

Using an Arrow Function

```
function App() {

   function handleClick(param, event) {

       console.log("Param:", param);

       console.log("Event Type:", event.type);

   }



   return (

       <button onClick={(event) ⇒ handleClick("Argument 3", event)}>

           Click Me

       </button>

   );

}
```

36. What are the common event handlers in React, and how are they used?

React provides a wide range of event handlers to handle user interactions. These event handlers are written in camelCase (e.g., `onClick`, `onChange`) and correspond to standard DOM events. Here’s a breakdown of the most commonly used event handlers and their usage:

Mouse Events

Event Handler	Description	Example Usage
`onClick`	Fired when an element is clicked	`<button onClick={handleClick}>Click Me</button>`
`onDoubleClick`	Fired when an element is double-clicked	`<div onDoubleClick={handleDoubleClick}>Double Click</div>`
`onMouseEnter`	Fired when the mouse enters an element	`<div onMouseEnter={handleMouseEnter}>Hover Here</div>`
`onMouseLeave`	Fired when the mouse leaves an element	`<div onMouseLeave={handleMouseLeave}>Hover Out</div>`
`onMouseMove`	Fired when the mouse moves over an element	`<div onMouseMove={handleMouseMove}>Move Mouse</div>`

Keyboard Events

Event Handler	Description	Example Usage
`onKeyDown`	Fired when a key is pressed down	`<input onKeyDown={handleKeyDown} />`
`onKeyPress`	Fired when a key is pressed (deprecated, use `onKeyDown`)	`<input onKeyPress={handleKeyPress} />`
`onKeyUp`	Fired when a key is released	`<input onKeyUp={handleKeyUp} />`

Form Events

Event Handler	Description	Example Usage
`onChange`	Fired when the value of an input element changes	`<input type="text" onChange={handleChange} />`
`onSubmit`	Fired when a form is submitted	`<form onSubmit={handleSubmit}>...</form>`
`onFocus`	Fired when an element gains focus	`<input type="text" onFocus={handleFocus} />`
`onBlur`	Fired when an element loses focus	`<input type="text" onBlur={handleBlur} />`

Touch Events

Event Handler	Description	Example Usage
`onTouchStart`	Fired when a touch starts on an element	`<div onTouchStart={handleTouchStart}>Touch Here</div>`
`onTouchMove`	Fired when a touch moves across an element	`<div onTouchMove={handleTouchMove}>Move Touch</div>`
`onTouchEnd`	Fired when a touch ends on an element	`<div onTouchEnd={handleTouchEnd}>Touch End</div>`

Focus Events

Event Handler	Description	Example Usage
`onFocus`	Fired when an element gains focus	`<input type="text" onFocus={handleFocus} />`
`onBlur`	Fired when an element loses focus	`<input type="text" onBlur={handleBlur} />`

React Router

37. What is React Router, and why is it used?

React Router is a popular library in the React ecosystem used for handling navigation and routing in a React application.

It enables you to build single-page applications (SPAs) with dynamic routing, allowing users to navigate between different views or pages without reloading the browser.

Core Concepts of React Router

Routes and Route Components:

a. Define the mapping between a URL path and the component to render.

```
import { BrowserRouter, Route, Routes } from "react-router-dom";


function App() {

   return (

       <BrowserRouter>

           <Routes>

               <Route path="/" element={<Home />} />

               <Route path="/about" element={<About />} />

           </Routes>

       </BrowserRouter>

   );

}
```

Router Types:

- `BrowserRouter`: Uses HTML5 history API (recommended for most applications).
- `HashRouter`: Uses hash fragments in URLs (e.g., `#/about`) and is useful for environments where server configuration is restricted.

Link Component:

```
import { Link } from "react-router-dom";



function Navigation() {

   return (
       <nav>
           <Link to="/">Home</Link>
           <Link to="/about">About</Link>
       </nav>
   );
}
```

Dynamic Routing (Route Parameters):

Allows routes to accept dynamic parameters.

```
import { useParams } from "react-router-dom";


function Profile() {
   const { userId } = useParams();
   return <h1>Profile of User {userId}</h1>;
}


// Routes definition
<Route path="/profile/:userId" element={<Profile />} />
```

Programmatic Navigation:

Use the `useNavigate` hook to navigate programmatically.

```
import { useNavigate } from "react-router-dom";


function Login() {

   const navigate = useNavigate();


   function handleLogin() {

       // Perform login logic

       navigate("/dashboard");

   }


   return <button onClick={handleLogin}>Login</button>;

}
```

Example Application Using React Router

App Structure

```
import { BrowserRouter, Routes, Route, Link } from "react-router-dom";


function App() {

   return (
       <BrowserRouter>

           <nav>

               <Link to="/">Home</Link>

               <Link to="/about">About</Link>

           </nav>



           <Routes>

               <Route path="/" element={<Home />} />

               <Route path="/about" element={<About />} />

           </Routes>

       </BrowserRouter>

   );

}



function Home() {

   return <h1>Welcome to the Home Page</h1>;

}



function About() {

   return <h1>About Us</h1>;

}
```

```
export default App;
```

38. How do you handle dynamic routing in React?

Dynamic routing in React allows you to create routes that can change based on parameters, user input, or data. This is particularly useful for applications with dynamic content, such as user profiles, blog posts, or product pages. React Router provides tools to handle dynamic routing effectively.

Here’s how to implement dynamic routing in React:

Define Routes with Parameters

Use a colon (`:`) to define a route parameter in the path.

```
import { BrowserRouter as Router, Routes, Route } from
"react-router-dom";


function App() {
 return (
   <Router>
     <Routes>
       <Route path="/user/:id" element={<UserProfile />} />
     </Routes>
   </Router>
 );

}
```

In this example, :id is a route parameter that can be dynamically replaced by any value, such as /user/123 or /user/john.

Access Route Parameters

Use the useParams hook provided by React Router to extract parameters from the URL.

```
import { useParams } from "react-router-dom";



function UserProfile() {

 const { id } = useParams(); // Extract the "id" parameter

 return <h1>Viewing profile of user: {id}</h1>;

}
```

If the URL is /user/123, id will be 123.

Handle Nested Routes

Dynamic routes can also be nested.

```
function App() {

   return (

     <Router>

       <Routes>

         <Route path="/user/:id" element={<UserProfile />}>
```

```
          <Route path="settings" element={<UserSettings />} />
        </Route>
      </Routes>
    </Router>
  );
}
```

To display nested routes, use an `<Outlet>` component in the parent component:

```
import { Outlet } from "react-router-dom";


function UserProfile() {
 const { id } = useParams();
 return (
   <div>
     <h1>User Profile: {id}</h1>
     <Outlet /> {/* Renders nested routes */}
   </div>
 );
}
```

Query Parameters

For query parameters like `/search?query=react`, use the `useSearchParams` hook:

```
import { useSearchParams } from "react-router-dom";


function Search() {

 const [searchParams] = useSearchParams();

 const query = searchParams.get("query"); // Extract "query" parameter

 return <h1>Search results for: {query}</h1>;

}
```

Redirect with Dynamic Parameters

Use the `useNavigate` hook to navigate programmatically with dynamic parameters:

```
import { useNavigate } from "react-router-dom";


function Dashboard() {

 const navigate = useNavigate();


 const goToProfile = (userId) ⇒ {

   navigate(`/user/${userId}`);
 };


 return (
   <button onClick={() => goToProfile(123)}>Go to User 123</button>
 );
}
```

Error Handling for Invalid Dynamic Routes

Define a "catch-all" route for unmatched paths or validate dynamic parameters:

```
import { Navigate } from "react-router-dom";


function App() {
 return (
   <Router>
     <Routes>
       <Route path="/user/:id" element={<UserProfile />} />
       <Route path="*" element={<Navigate to="/" />} /> {/* Redirect to
home */}
     </Routes>
```

```
    </Router>
  );
}


// Validation inside the component
function UserProfile() {
  const { id } = useParams();
  if (!isValidId(id)) {
    return <h1>Invalid User ID</h1>;
  }
  return <h1>User Profile: {id}</h1>;
}
```

Forms in React

43. How do you handle forms in React?

Handling forms in React involves managing the form's state and responding to user inputs. React's component-based architecture makes it easy to build interactive forms with controlled or uncontrolled components.

Controlled Components

In controlled components, the form inputs are controlled by React state. Each input field's value is tied to a state variable.

Example: A Simple Controlled Form

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


function ControlledForm() {
 const [formData, setFormData] = useState({
   name: "",
   email: "",
   message: "",
 });


 const handleChange = (e) ⇒ {
   const { name, value } = e.target;
   setFormData({ ...formData, [name]: value });
 };


 const handleSubmit = (e) ⇒ {
   e.preventDefault();

   console.log("Form submitted:", formData);

   // Perform actions like API calls or validation

 };


 return (

   <form onSubmit={handleSubmit}>

     <div>

       <label>

         Name:

         <input

           type="text"

           name="name"

           value={formData.name}

           onChange={handleChange}

         />

       </label>

     </div>

     <div>

       <label>

         Email:
```

```
        <input
          type="email"
          name="email"
          value={formData.email}
          onChange={handleChange}
        />
      </label>
    </div>
    <div>
      <label>
        Message:
        <textarea
          name="message"
          value={formData.message}
          onChange={handleChange}
        />
      </label>
    </div>
    <button type="submit">Submit</button>
  </form>
);
```

```
}


export default ControlledForm;
```

Uncontrolled Components

In uncontrolled components, form inputs use the DOM's default behavior, and you retrieve their values using `ref`.

Example: Uncontrolled Form

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


function UncontrolledForm() {
 const nameRef = useRef();
 const emailRef = useRef();


 const handleSubmit = (e) ⇒ {
   e.preventDefault();
   const name = nameRef.current.value;
   const email = emailRef.current.value;
   console.log("Form submitted:", { name, email });
 };


 return (
   <form onSubmit={handleSubmit}>

     <div>

       <label>

         Name:

         <input type="text" ref={nameRef} />

       </label>

     </div>

     <div>

       <label>

         Email:

         <input type="email" ref={emailRef} />

       </label>

     </div>

     <button type="submit">Submit</button>

   </form>

 );

}



export default UncontrolledForm;
```

Validating Form Inputs

You can validate inputs either during user input (`onChange`), on blur (`onBlur`), or on form submission.

```
function FormWithValidation() {

   const [email, setEmail] = useState("");

   const [error, setError] = useState("");

    const handleChange = (e) ⇒ {

     const value = e.target.value;

     setEmail(value);

     if (!/\S+@\S+\.\S+/.test(value)) {

       setError("Invalid email address");

     } else {

       setError("");

     }

   };

    const handleSubmit = (e) ⇒ {

     e.preventDefault();

     if (!error) {

       console.log("Valid email:", email);

     }

   };

    return (

     <form onSubmit={handleSubmit}>
```

```
      <div>
        <label>
          Email:
          <input type="email" value={email} onChange={handleChange} />
        </label>
        {error && <p style={{ color: "red" }}>{error}</p>}
      </div>
      <button type="submit" disabled={!!error}>
        Submit
      </button>
    </form>
  );
}
```

Handling Complex Forms

For larger forms with many fields, consider using libraries like **Formik**, **React Hook Form**, or **Yup** for validation.

44. How do you manage form validation in React?

Managing form validation in React involves ensuring that user inputs meet specific requirements before processing or submitting data. React provides flexibility in implementing validation, and you can choose between manual validation, custom logic, or third-party libraries.

Choosing the Right Approach

45. **Simple Forms**: Use manual or live validation.
46. **Complex Forms**: Use libraries like **Formik** or **React Hook Form** for better scalability and maintainability.
47. **Schema-Based Validation**: Combine Formik or React Hook Form with Yup for robust validation rules.

48. How do you integrate libraries like Formik or React Hook Form for form handling?

Integrating libraries like **Formik** or **React Hook Form** into your React application simplifies form handling and validation. Here's a step-by-step guide for using each library:

Formik Integration

Formik is a popular library for managing form state, validation, and submission in React applications.

```
npm install formik yup
```

Basic Usage

Formik provides components like `Formik`, `Form`, `Field`, and `ErrorMessage` to handle form operations.

```
import React from "react";

import { Formik, Form, Field, ErrorMessage } from "formik";



function FormikExample() {
```

```
return (
  <Formik
    initialValues={{ name: "", email: "" }} // Initial form values
    validate={(values) => {
      const errors = {};
      if (!values.name) {
        errors.name = "Name is required";
      }
      if (!values.email) {
        errors.email = "Email is required";
      } else if (!/\S+@\S+\.\S+/.test(values.email)) {
        errors.email = "Invalid email address";
      }
      return errors;
    }}
    onSubmit={(values, { setSubmitting }) => {
      console.log("Form submitted:", values);
      setSubmitting(false); // Reset submitting state
    }}
  >
    {({ isSubmitting }) => (
      <Form>
        <div>
          <label>
            Name:
            <Field type="text" name="name" />
          </label>
          <ErrorMessage name="name" component="p" style={{ color: "red"
}} />
        </div>
        <div>
          <label>
            Email:
            <Field type="email" name="email" />
          </label>
          <ErrorMessage name="email" component="p" style={{ color:
"red" }} />
        </div>
        <button type="submit" disabled={isSubmitting}>
          Submit
        </button>
      </Form>
    )}
```

```
    </Formik>
  );
}

export default FormikExample;
```

Using Yup for Validation

Yup is a schema validation library that integrates well with Formik.

Example: Using Yup

```
import React from "react";
import { Formik, Form, Field, ErrorMessage } from "formik";
import * as Yup from "yup";

const validationSchema = Yup.object({
 name: Yup.string().required("Name is required"),
 email: Yup.string()
   .email("Invalid email address")
   .required("Email is required"),
});
```

```
function FormikWithYup() {

 return (

   <Formik

     initialValues={{ name: "", email: "" }}

     validationSchema={validationSchema}

     onSubmit={(values) => {

       console.log("Form submitted:", values);

     }}

   >

     <Form>

       <div>

         <label>

           Name:

           <Field type="text" name="name" />

         </label>

         <ErrorMessage name="name" component="p" style={{ color: "red"
}} />

       </div>

       <div>

         <label>

           Email:

           <Field type="email" name="email" />
         </label>

         <ErrorMessage name="email" component="p" style={{ color: "red" }} />

       </div>

       <button type="submit">Submit</button>

     </Form>

   </Formik>

 );

}


export default FormikWithYup;
```

Integrating with Yup for Validation

React Hook Form supports schema-based validation with Yup.

Example: Using Yup with React Hook Form

```
import React from "react";

import { useForm } from "react-hook-form";

import { yupResolver } from "@hookform/resolvers/yup";

import * as Yup from "yup";


const validationSchema = Yup.object({

 name: Yup.string().required("Name is required"),
 email: Yup.string()
   .email("Invalid email address")
   .required("Email is required"),
});

function ReactHookFormWithYup() {
 const {
   register,
   handleSubmit,
   formState: { errors },
 } = useForm({
   resolver: yupResolver(validationSchema),
 });

 const onSubmit = (data) ⇒ {
   console.log("Form Data:", data);
 };

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

```
        <label>
          Name:
          <input {...register("name")} />
        </label>
        {errors.name && <p style={{ color: "red"
}}>{errors.name.message}</p>}
      </div>
      <div>
        <label>
          Email:
          <input {...register("email")} />
        </label>
        {errors.email && <p style={{ color: "red"
}}>{errors.email.message}</p>}
      </div>
      <button type="submit">Submit</button>
    </form>
  );
}

export default ReactHookFormWithYup;
```

State Management

48. What is prop drilling, and how do you solve it?

Prop drilling refers to the process of passing data from a parent component to deeply nested child components in a React application through multiple intermediate components.

It can become problematic as the application grows, making the code harder to maintain, debug, and refactor.

Example of Prop Drilling

Imagine you have the following component structure:

`Parent → Child → Grandchild → GreatGrandchild`

If the `Parent` component has some data or functions needed by the `GreatGrandchild`, you would need to pass these props down through every intermediate component (Child and Grandchild), even if they don't use them directly.

```
function Parent() {
   const data = "Hello from Parent";
    return <Child data={data} />;
 }
  function Child({ data }) {
   return <Grandchild data={data} />;
 }
  function Grandchild({ data }) {
   return <GreatGrandchild data={data} />;
 }
  function GreatGrandchild({ data }) {
   return <div>{data}</div>;
 }
```

Problems with Prop Drilling

1. **Cluttered Code**: Components that don't use the props still have to handle and pass them down.
2. **Tight Coupling**: Changes in the props structure require updates across multiple components.
3. **Reduced Reusability**: Components become less reusable since they depend on specific props being passed.

Solutions to Prop Drilling

React Context API

a. **What it does**: Provides a way to share values (like data or functions) between components without explicitly passing props.
b. **How it works**:
   i. Create a `Context` object using `React.createContext`.
   ii. Use a `Provider` to supply the value.
   iii. Access the value with `useContext` in the child components.
c. **Example**:

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

```
const DataContext = createContext();


function Parent() {
 const data = "Hello from Parent";


 return (
   <DataContext.Provider value={data}>
     <Child />
   </DataContext.Provider>
 );
}


function GreatGrandchild() {
 const data = useContext(DataContext);
 return <div>{data}</div>;
}
```

State Management Libraries

- **Libraries**: Redux, MobX, Zustand, Recoil, etc.
- **What they do**: Manage global application state, making it accessible across components without prop drilling.
- **When to use**: For larger applications with complex state management needs.

Component Composition

Pass only the necessary functions or components down the tree.

Example:

```
function Parent() {
   return <Child render={(data) ⇒ <GreatGrandchild data={data} />} />;
 }
  function Child({ render }) {
   const data = "Hello from Parent";
   return render(data);
 }
  function GreatGrandchild({ data }) {
   return <div>{data}</div>;
 }
```

Hooks (Custom or Built-In)

Create custom hooks to encapsulate shared logic.

Example:

```
const useData = () ⇒ "Hello from Hook";


function GreatGrandchild() {
 const data = useData();
 return <div>{data}</div>;
}
```

Choosing the Right Solution

49. For **small applications**: React Context API is sufficient.
50. For **medium to large applications**: State management libraries like Redux or Zustand provide better scalability.
51. For **specific cases**: Custom hooks or component composition might be sufficient.

52. What are the common state management libraries used in React?

State management is crucial in React applications, especially as they grow in complexity. While React has a built-in state management system (using `useState`, `useReducer`, and Context API), developers often rely on external libraries for more robust and scalable solutions.

Here are some of the most common state management libraries used in React:

1. Redux

- **Overview**: Redux is one of the most popular state management libraries in React. It follows a predictable state container approach using a unidirectional data flow.
- **Key Features**:
  - Centralized state management.
  - Strict unidirectional data flow.
  - Middleware support for asynchronous actions (e.g., `redux-thunk`, `redux-saga`).
- **When to Use**:
  - Large-scale applications with complex state.
  - Need for debugging tools (Redux DevTools).
- **Cons**:
  - Boilerplate-heavy, though this has improved with the `Redux Toolkit`.

- **Website**: Redux

---

2. MobX

- **Overview**: MobX is a library that emphasizes simplicity and reactivity. It uses observable state and automatically tracks dependencies.
- **Key Features**:
  - Observable state and automatic updates.
  - Minimal boilerplate.
  - Fine-grained reactivity (updates only the components that depend on the changed state).
- **When to Use**:
  - Applications needing reactive programming.
  - When simplicity and less boilerplate are priorities.
- **Cons**:
  - Learning curve for observables.
- **Website**: MobX

---

3. Zustand

- **Overview**: A lightweight and simple state management library built on hooks.
- **Key Features**:
  - Minimalistic (only a few kilobytes in size).
  - Store-based with a hook-driven API.
  - Reactivity through shallow comparison.
- **When to Use**:
  - Small to medium applications.
  - Scenarios requiring lightweight state management.
- **Cons**:
  - Limited ecosystem compared to Redux or MobX.
- **Website**: Zustand

4. Recoil

- **Overview**: A state management library from Facebook that focuses on simplicity and performance, designed specifically for React.
- **Key Features**:
    - Atom-based state (small pieces of state that components subscribe to).
    - Read-only selectors for derived state.
    - Concurrent mode support.
- **When to Use**:
    - Applications requiring fine-grained state management.
    - When React Concurrent Mode compatibility is needed.
- **Cons**:
    - Still evolving; not as mature as Redux or MobX.
- **Website**: Recoil

---

5. Jotai

- **Overview**: A minimalistic state management library that emphasizes atomic state and simplicity.
- **Key Features**:
    - Atom-based state management.
    - Zero boilerplate.
    - Built-in support for derived and asynchronous state.
- **When to Use**:
    - Small to medium applications needing atomic state.
- **Cons**:
    - Smaller community compared to larger libraries.
- **Website**: Jotai

---

6. React Query (TanStack Query)

- **Overview**: Primarily used for managing server-state (e.g., API data fetching and caching).
- **Key Features**:
    - Server-state management with caching and synchronization.
    - Automatic background refetching.
    - Optimistic updates for a better UX.
- **When to Use**:
    - Applications with significant server-state (e.g., APIs, databases).
    - When data-fetching needs are complex.
- **Cons**:
    - Not a full state management library; focuses on server-state.
- **Website**: React Query

---

7. XState

- **Overview**: A state management library based on finite state machines and statecharts.
- **Key Features**:
    - Explicit state transitions.
    - Visual debugging tools.
    - Handles complex workflows and logic-heavy applications.
- **When to Use**:
    - Applications with complex workflows or finite state logic.
- **Cons**:
    - Requires a shift in thinking for developers not familiar with state machines.
- **Website**: XState

---

8. Apollo Client

Overview: A GraphQL client that also manages client-side state in addition to server-state.

Key Features:

a. Manages both server and local state using GraphQL.
b. Caching and real-time updates.

When to Use:

c. Applications that rely heavily on GraphQL APIs.

Cons:

d. Overhead if not using GraphQL.

Website: Apollo Client

Summary Table

Library	Size	Complexity	Best For
Redux	Medium	High	Large-scale applications.
MobX	Medium	Low	Reactive state management.
Zustand	Small	Low	Lightweight state management.
Recoil	Medium	Medium	React-specific state, concurrent mode.
Jotai	Small	Low	Atomic and minimalistic state.
React Query	Medium	Low	Server-state management.
XState	Medium	High	Complex workflows or state logic.
Apollo Client	Large	High	GraphQL-centric applications.

53. How do you integrate Redux with React?

54. What is the difference between Redux and the Context API?

Both **Redux** and the **React Context API** are used for state management in React applications, but they serve different purposes and have distinct characteristics. Here's a detailed comparison:

---

1. Purpose

- **Redux**:
  - A dedicated state management library with a predictable state container.
  - Ideal for managing complex application-wide state and interactions between components.
  - Comes with advanced tools for debugging, middleware for asynchronous actions, and scalability for large applications.
- **Context API**:
  - A React feature for sharing data across component trees without prop drilling.
  - Best suited for lightweight state management, such as themes, user authentication, or global configuration.

---

2. Learning Curve

- **Redux**:
  - Steeper learning curve due to concepts like actions, reducers, middleware, and the unidirectional data flow.
  - Requires understanding of the Redux Toolkit (modern Redux) for reducing boilerplate.
- **Context API**:
  - Built into React and straightforward to learn.
  - Simpler to set up and use compared to Redux.

---

3. Boilerplate

- **Redux**:

  - Historically known for being verbose, with separate files for actions, reducers, and store configuration.
  - The Redux Toolkit significantly reduces boilerplate but still requires more setup than Context API.
- **Context API**:
  - Minimal boilerplate, as it directly integrates with React through `createContext`, `Provider`, and `useContext`.

---

4. State Management Complexity

- **Redux**:
  - Designed for managing complex, large-scale state.
  - Handles derived state, asynchronous logic, and data normalization better.
  - Supports middleware (e.g., `redux-thunk` or `redux-saga`) for handling side effects.
- **Context API**:
  - Suitable for simpler state-sharing needs.
  - Not ideal for highly dynamic or deeply nested state, as frequent updates can lead to performance issues due to re-renders.

---

5. Performance

- **Redux**:
  - More optimized for performance in large applications.
  - Components can subscribe to specific slices of state, reducing unnecessary re-renders.
- **Context API**:

    - Every update to the context value triggers a re-render of all components consuming that context.
    - This can be mitigated with techniques like splitting context or memoization, but it requires careful handling.

---

6. Ecosystem and Debugging

- **Redux**:
    - A rich ecosystem with tools like Redux DevTools for time-travel debugging and state inspection.
    - Middleware and plugins for handling asynchronous operations, API calls, and more.
- **Context API**:
    - No built-in debugging tools.
    - Relies on standard React DevTools.

---

7. Scalability

- **Redux**:
    - Built for scalability; ideal for applications with extensive state and multiple developers.
    - Easily extendable and maintainable due to its clear structure.
- **Context API**:
    - Works well for smaller applications or isolated use cases (e.g., theming or auth).
    - Can become unwieldy in large applications due to the lack of structure and performance issues.

---

8. Asynchronous Logic

- **Redux**:
    - Handles asynchronous operations using middleware like `redux-thunk` or `redux-saga`.
    - Provides a clear mechanism for managing side effects.
- **Context API**:
    - No built-in support for handling asynchronous actions.
    - Asynchronous logic needs to be managed manually (e.g., within custom hooks).

When to Use Which

Criterion	Redux	Context API
Application size	Large, complex applications.	Small to medium applications.
State complexity	Complex, interdependent state.	Simple, isolated state-sharing needs.
Performance sensitivity	High performance requirements.	Less sensitive to performance.
Developer team	Multiple developers, scalability needed.	Single developer or small teams.

Ease of setup	Moderate setup required.	Minimal setup required.
Debugging	Advanced debugging with DevTools.	Basic debugging using React DevTools.

Summary

- Use **Redux** for large applications where state complexity, scalability, and performance optimization are priorities.
- Use the **Context API** for simpler use cases, such as theming, authentication, or small apps where Redux might feel like overkill.

In some scenarios, a combination of both might be appropriate: use Context API for lightweight global state and Redux for managing complex application logic.

Rendering and Performance

53. What are keys in React? Why are they important in lists?

In React, **keys** are unique identifiers assigned to elements in a list or collection of components. They help React identify which items have changed, been added, or been removed, and thus, optimize rendering.

Importance of Keys in Lists:

1. **Efficient Rendering:** React uses keys to track which elements have changed in the DOM. When the list changes (such as adding, removing, or reordering items), React can efficiently update only the elements that have actually changed. Without keys, React would have to re-render the entire list, which can be inefficient.

2. **Preserving Component State:** If the list elements are dynamic components (e.g., a list of input fields), keys help React maintain the internal state of each component, even when the list is re-ordered or modified. Without keys, the state might get mixed up because React could reuse components in the wrong order.
3. **Identifying Changes in Dynamic Lists:** When data changes, React uses the key to match the updated data with the previously rendered elements, minimizing the number of changes in the DOM.

How to Use Keys:

54. Keys should be **unique** among siblings but don't have to be globally unique. Commonly, you use an ID from your data as the key.

```
const items = ['apple', 'banana', 'cherry'];


function List() {

 return (

   <ul>

     {items.map((item, index) => (

       <li key={item}>{item}</li>  // Unique key for each list item

     ))}

   </ul>

 );

}
```

In this example, `item` (the name of the fruit) is used as the key because it's unique in this list. If you have a more complex list, it's common to use an ID from your data structure, for example, `key={item.id}`.

Key Takeaways:

- Keys help React identify and optimize updates.
- They must be **unique within the same list**.
- They improve performance by minimizing re-renders and preserving component state.

55. What is conditional rendering? Provide an example.

Conditional rendering in React refers to rendering different UI elements or components based on certain conditions. It allows you to display content dynamically depending on the state or props of the component.

In React, you can implement conditional rendering using JavaScript operators like:

56. **If-else statements**
57. **Ternary operators**
58. **Logical AND (`&&`) operator**

Example 1: Using `if-else` Statements

```
function Welcome({ isLoggedIn }) {

   if (isLoggedIn) {

     return <h1>Welcome back!</h1>;

   } else {

     return <h1>Please log in.</h1>;

   }

 }

  export default Welcome;
```

In this example, based on the isLoggedIn prop, the component either renders "Welcome back!" or "Please log in."

Example 2: Using Ternary Operator

A more concise way to handle conditional rendering is by using the ternary operator:

```
function Welcome({ isLoggedIn }) {

   return (

     <h1>{isLoggedIn ? 'Welcome back!' : 'Please log in.'}</h1>

   );

 }

  export default Welcome;
```

Here, if isLoggedIn is true, it renders "Welcome back!"; otherwise, it renders "Please log in."

Example 3: Using Logical AND (&&) Operator

You can also use the && operator for rendering content conditionally, especially when you want to render an element only when a certain condition is true.

```
function Message({ isLoggedIn }) {

   return (

     <div>

       {isLoggedIn && <p>You are logged in!</p>}

     </div>

   );

 }
```

```
  export default Message;
```

In this example, the paragraph with the message "You are logged in!" will only be displayed if isLoggedIn is true.

Key Takeaways:

59. Conditional rendering is useful when you need to display content based on certain conditions.
60. You can use if-else, ternary operators, or logical operators for conditional rendering.
61. It's a common pattern for things like showing login forms, displaying different views, or handling user permissions.

62. What is React.memo, and how does it work?

`React.memo` is a higher-order component in React that helps optimize the performance of functional components by memoizing them. It prevents unnecessary re-renders by only re-rendering the component when its props change.

`React.memo` is used to wrap a component and makes React compare the current props with the previous props. If the props haven't changed, the component is **not re-rendered**, and React reuses the previous rendered output.

This is particularly useful when:

- The component receives the same props and doesn't need to re-render.
- You want to prevent re-rendering of a child component when its parent re-renders.

Syntax:

```
const MyComponent = React.memo(function MyComponent(props) {

  // Component logic here

  return <div>{props.name}</div>;
```

```
 });
```

In this example, `MyComponent` will only re-render if its `props.name` value changes. If the `name` prop remains the same, React will skip rendering the component and use the cached result.

Example:

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


const Greeting = React.memo(({ name }) ⇒ {

 console.log("Rendering Greeting Component");

 return <h1>Hello, {name}!</h1>;

});


function App() {

 const [name, setName] = useState("Alice");

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


 return (

   <div>

     <Greeting name={name} />
```

```
      <button onClick={() => setCount(count + 1)}>Increase count:
{count}</button>
      <button onClick={() => setName(name === "Alice" ? "Bob" :
"Alice")}>Change name</button>
    </div>
  );
}

export default App;
```

In this example:

- The Greeting component is wrapped in React.memo.
- Clicking the "Increase count" button will **not** trigger a re-render of Greeting because the name prop hasn't changed. Only the state variable count changes.
- Clicking the "Change name" button will re-render Greeting because the name prop changes.

Custom Comparison Function:

By default, React.memo performs a shallow comparison of the props. However, you can provide a **custom comparison function** if you need more control over how props are compared.

```
const Greeting = React.memo(
    ({ name, age }) => {
      console.log("Rendering Greeting Component");
      return <h1>Hello, {name}, Age: {age}!</h1>;
    },
    (prevProps, nextProps) ⇒ {
      // Only re-render if 'name' changes
      return prevProps.name === nextProps.name;
    }
  );
```

In this case, `Greeting` will only re-render if the `name` prop changes, regardless of changes to `age`.

Key Takeaways:

63. **`React.memo`** memoizes functional components to prevent unnecessary re-renders.
64. It only re-renders a component when its props change.
65. You can optionally provide a custom comparison function to control when re-renders occur.
66. This is especially useful for performance optimization in large applications or when dealing with frequently re-rendered components.

67. What is lazy loading in React? How do you implement it?

Lazy loading in React is a technique that allows you to load components only when they are needed, instead of loading them all at once at the initial render.

This can significantly improve the performance of your application, particularly for larger applications with many components.

In React, **lazy loading** is achieved using the `React.lazy()` function, which allows you to define components that will only be loaded when they're rendered.

`React.lazy()` allows you to dynamically import a component using **code splitting**. When you use `React.lazy()`, React will automatically load the component only when it's needed (i.e., when it's rendered for the first time).

This helps in reducing the initial bundle size, leading to faster page loads.

Syntax:

```
const MyComponent = React.lazy(() ⇒ import('./MyComponent'));
```

Here, `import('./MyComponent')` dynamically imports the `MyComponent` only when it's actually rendered.

Using Suspense:

`Suspense` is a built-in React component that lets you define what should be displayed while the lazy-loaded component is being fetched. It's like a loading state for dynamically imported components.

Basic Example of Lazy Loading:

Create a component (`MyComponent.js`):

```
// MyComponent.js

import React from 'react';


function MyComponent() {

 return <h1>This is a lazy-loaded component!</h1>;

}
```

```
export default MyComponent;
```

Use `React.lazy()` and `Suspense` in the parent component:

```
// App.js

import React, { Suspense } from 'react';


// Lazy load the MyComponent

const MyComponent = React.lazy(() ⇒ import('./MyComponent'));


function App() {

 return (

   <div>

     <h1>Main App</h1>

     {/* Suspense is required to show a fallback while the component is
being loaded */}

     <Suspense fallback={<div>Loading ... </div>}>

       <MyComponent />

     </Suspense>

   </div>

 );

}
```

```
export default App;
```

Lazy loading improves the performance of large React applications by splitting the code and loading only the necessary components when needed.

`React.lazy()` is used to dynamically import components.

`Suspense` is used to show a loading state while the component is being loaded.

Lazy loading is particularly useful for routes or components that are not immediately required, allowing for faster initial rendering of the application.

Debugging

58. What are some common errors in React applications and how do you debug them?
59. How do you use React Developer Tools in a browser?
60. What is the purpose of error boundaries in React?
61. How do you handle errors in asynchronous operations in React?

React Ecosystem

62. What is ReactDOM, and how is it different from React?
63. What are fragments in React? Why are they used?

In React, **fragments** are a lightweight way to group a list of child elements without adding extra nodes to the DOM.

They were introduced to address scenarios where multiple child elements need to be returned from a component but wrapping them in an unnecessary `<div>` or another container would create unwanted DOM elements.

A fragment is represented using the `<React.Fragment>` component or its shorthand syntax (`<> </>`). It does not produce any additional DOM nodes.

Example:

```
function Example() {
   return (
     <React.Fragment>
       <h1>Heading</h1>
       <p>This is a paragraph.</p>
     </React.Fragment>
   );
 }
  // Shorthand syntax
 function ExampleShort() {
   return (
     <>
       <h1>Heading</h1>
       <p>This is a paragraph.</p>
     </>
   );
 }
```

In both cases, the rendered output in the DOM will only contain the `<h1>` and `<p>` elements without an additional wrapper.

Why Use Fragments?

64. **Avoid Unnecessary DOM Nodes**
    a. Adding unnecessary wrapper elements (like `<div>`) can bloat the DOM, which may cause issues with styling, layout, or performance.
    b. Example problem without fragments:

```
function WithoutFragments() {

   return (

     <div>

       <h1>Heading</h1>

       <p>Text</p>

     </div>

   );

 }
```

- The extra `<div>` may interfere with CSS or lead to a less semantic structure.

2. **Performance Optimization**
    - Fragments avoid the cost of creating, managing, and rendering additional DOM nodes.
    - They are especially useful in large or deeply nested component trees.
3. **Simplify Code Structure**
    - Using fragments helps keep the code clean and focused, especially when rendering lists or conditional JSX elements.
4. **Return Multiple Children from Components**
    - React components must return a single parent element. Fragments allow grouping multiple child elements without adding an unnecessary wrapper.

When to Use Fragments

Rendering Lists: When rendering multiple list items without needing a wrapper element

```
function List() {

   return (

     <>

       <li>Item 1</li>

       <li>Item 2</li>

       <li>Item 3</li>

     </>

   );

 }
```

Layout Elements: Grouping child elements in a layout without disturbing the structure.

```
function Layout() {

   return (

     <>

       <header>Header</header>

       <main>Content</main>

       <footer>Footer</footer>

     </>

   );
}
```

When Parent Element is Not Desired: To avoid DOM pollution in scenarios like table rows, where extra nodes can break the structure.

```
function Table() {
  return (
    <table>
      <tbody>
        <>
          <tr>
            <td>Row 1, Col 1</td>
            <td>Row 1, Col 2</td>
          </tr>
          <tr>
            <td>Row 2, Col 1</td>
            <td>Row 2, Col 2</td>
          </tr>
        </>
      </tbody>
    </table>
  );
}
```

Limitations of Fragments

No Attributes: Standard fragments (<React.Fragment> or <>) cannot accept attributes like className or id. If you need attributes, you must use the long-form <React.Fragment> and avoid the shorthand.

```
<React.Fragment key="uniqueKey">

 <ChildComponent />

</React.Fragment>
```

1. **Not Always Needed**: In simpler scenarios, adding a parent element like <div> is sufficient and doesn't harm performance significantly.

Key Takeaways

Fragments are a clean, efficient way to return multiple children without extra DOM elements.

Use them to simplify your JSX structure and avoid unnecessary wrappers, improving performance and reducing DOM clutter.

They are especially useful when working with lists, layouts, and components that render multiple elements.

65. What is the significance of React.StrictMode?
66. What is the role of React.Children in React applications?

API Integration

66. How do you fetch data in React? Provide an example.

In React, you can fetch data using the `fetch` API or libraries like `axios`. The most common approach is to use the `useEffect` hook to trigger data fetching when a component mounts or updates

Example: Fetching Data with `fetch` and `useEffect`

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


const DataFetchingComponent = () ⇒ {

 const [data, setData] = useState([]);

 const [loading, setLoading] = useState(true);

 const [error, setError] = useState(null);


 useEffect(() ⇒ {

   const fetchData = async () ⇒ {

     try {

       const response = await
fetch("https://jsonplaceholder.typicode.com/posts");

       if (!response.ok) {

         throw new Error("Network response was not ok");

       }

       const result = await response.json();

       setData(result);

     } catch (error) {
       setError(error.message);
     } finally {
       setLoading(false);
     }
   };

   fetchData();
 }, []); // Empty dependency array ensures this runs only once after the
component mounts.

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

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

```
 );

};


export default DataFetchingComponent;
```

67. How do you handle loading and error states in API calls?

Handling **loading** and **error states** effectively in API calls ensures a smooth user experience and makes your application more robust. Here's a detailed explanation with an example:

Steps to Handle Loading and Error States:

1. Define States:

- Use `useState` to manage:
  - **Loading state**: Indicates whether the API call is in progress.
  - **Error state**: Captures and displays any errors from the API call.
  - **Data state**: Stores the fetched data.

Example:

```
const [data, setData] = useState(null);

const [loading, setLoading] = useState(true);

const [error, setError] = useState(null);
```

Set Initial States:

- When the API call starts:
  - Set `loading` to `true`.
  - Reset `error` to `null`.

Example:

```
useEffect(() ⇒ {

   const fetchData = async () ⇒ {

     setLoading(true);

     setError(null);

     try {

       const response = await fetch("https://api.example.com/data");

       if (!response.ok) {

         throw new Error("Failed to fetch data");

       }

       const result = await response.json();

       setData(result);

     } catch (err) {

       setError(err.message);

     } finally {

       setLoading(false); // Always set loading to false when done

     }

   };
```

```
    fetchData();

 }, []);
```

68. How do you handle pagination with APIs in React?

Handling pagination with APIs in React involves fetching a specific subset of data (e.g., a page) from the API and allowing users to navigate through the pages. Here's how you can implement pagination step by step:

Steps to Handle Pagination in React

1. API Design for Pagination

Ensure the API supports pagination. Common patterns include:

- **Query Parameters**: Use parameters like `page` and `limit`. Example: `https://api.example.com/items?page=1&limit=10`
- **Cursor-Based Pagination**: Use `cursor` tokens instead of page numbers. Example: `https://api.example.com/items?cursor=abc123`

---

2. Manage Pagination State

Create state variables for:

- Current page number.
- Data for the current page.
- Total pages or items (if provided by the API).

3. Fetch Data Based on Page

Trigger API calls whenever the page changes.

4. Display Pagination Controls

Provide buttons or links to navigate between pages.

Code Example: Pagination with Page and Limit

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


const PaginatedComponent = () ⇒ {

 const [data, setData] = useState([]);

 const [loading, setLoading] = useState(true);

 const [error, setError] = useState(null);

 const [currentPage, setCurrentPage] = useState(1);

 const [totalPages, setTotalPages] = useState(0);


 const itemsPerPage = 10;


 useEffect(() ⇒ {

   const fetchData = async () ⇒ {

     setLoading(true);

     setError(null);

     try {

       const response = await fetch(
`https://jsonplaceholder.typicode.com/posts?_page=${currentPage}&_limit=
${itemsPerPage}`
       );
       if (!response.ok) {
         throw new Error("Failed to fetch data");
       }

       const totalItems = response.headers.get("X-Total-Count");
       setTotalPages(Math.ceil(totalItems / itemsPerPage));
       const result = await response.json();
       setData(result);
     } catch (err) {
       setError(err.message);
     } finally {
       setLoading(false);
     }
   };

   fetchData();
 }, [currentPage]); // Refetch data whenever currentPage changes
```

```
const handleNextPage = () => {

  if (currentPage < totalPages) {

    setCurrentPage((prevPage) => prevPage + 1);

  }

};



const handlePreviousPage = () => {

  if (currentPage > 1) {

    setCurrentPage((prevPage) => prevPage - 1);

  }

};



if (loading) return <p>Loading...</p>;

if (error) return <p>Error: {error}</p>;



return (

  <div>

    <h1>Paginated Posts</h1>

    <ul>

      {data.map((item) => (

        <li key={item.id}>{item.title}</li>
```

```
       ))}

     </ul>

     <div>

       <button onClick={handlePreviousPage} disabled={currentPage ===
1}>

         Previous

       </button>

       <span>

         Page {currentPage} of {totalPages}

       </span>

       <button onClick={handleNextPage} disabled={currentPage ===
totalPages}>

         Next

       </button>

     </div>

   </div>

 );

};


export default PaginatedComponent;
```

Explanation:

69. **Pagination State**:

a. `currentPage`: Tracks the active page.

b. `totalPages`: Total number of pages (calculated from the `X-Total-Count` header).

70. **API Call**:

a. Fetches data for the current page using the `_page` and `_limit` query parameters.

71. **Pagination Controls**:

a. "Previous" and "Next" buttons to navigate between pages.

b. `disabled` property to disable buttons on boundary conditions.

72. **Dynamic Dependency**:

a. The `useEffect` hook re-fetches data whenever `currentPage` changes.

73. What is the difference between Axios and Fetch API?

Both **Axios** and the **Fetch API** are popular tools for making HTTP requests in JavaScript. While they can accomplish the same tasks, they differ in functionality, usability, and features. Here's a detailed comparison:

Feature	Fetch API	Axios
Ease of Use	More verbose; manual JSON parsing required.	Simpler syntax; JSON parsed by default.
Error Handling	Only rejects on network errors.	Rejects on HTTP errors (e.g., 404, 500).
Default Timeouts	Not supported natively.	Can set timeouts easily.
Request/Response Interceptors	Not supported.	Supported for preprocessing requests or responses.
URL Query Parameters	Manual concatenation needed.	Built-in support with `params` object.
Cross-browser Compatibility	Supported in modern browsers.	Works in older browsers with polyfills.

Testing React Applications

71. How do you test React components?
72. What is the difference between shallow rendering and full DOM rendering?
73. What is Jest, and how is it used with React?
74. How do you test hooks in React?

Testing React hooks ensures that they behave as expected under different conditions. Here's how you can effectively test hooks:

---

1. Testing with a Component

Hooks like `useState` or `useEffect` need to run inside a functional React component. You can create a test component to render and interact with the hook.

Example: Testing `useState`

```
import { render, screen, fireEvent } from '@testing-library/react';

import React, { useState } from 'react';



function TestComponent() {

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



 return (

   <div>

     <p>Count: {count}</p>

     <button onClick={() ⇒ setCount((prev) ⇒ prev +
1)}>Increment</button>
```

```
    </div>
  );
}


test('increments counter', () => {
 render(<TestComponent />);
  const button = screen.getByText('Increment');
 fireEvent.click(button);
 expect(screen.getByText('Count: 1')).toBeInTheDocument();
});
```

---

2. Using `@testing-library/react-hooks`

The `@testing-library/react-hooks` library provides utilities specifically designed to test hooks directly.

Installation

```
npm install @testing-library/react-hooks
```

Example: Testing a Custom Hook

```
import { renderHook, act } from '@testing-library/react-hooks';


// Custom hook

function useCounter() {

 const [count, setCount] = React.useState(0);

 const increment = () ⇒ setCount((prev) ⇒ prev + 1);

 return { count, increment };

}


test('should increment counter', () ⇒ {

 const { result } = renderHook(() ⇒ useCounter());


 // Initial state

 expect(result.current.count).toBe(0);


 // Increment counter

 act(() ⇒ {

   result.current.increment();

 });


 expect(result.current.count).toBe(1);
});
```

3. Mocking API Calls in Hooks

If your hook involves API calls, use mocking libraries like `jest.fn()` or `msw`.

Example: Testing `useEffect` with API Calls

```
import { renderHook } from '@testing-library/react-hooks';

import { act } from 'react-dom/test-utils';

import axios from 'axios';



// Mock axios

jest.mock('axios');

const mockedAxios = axios as jest.Mocked<typeof axios>;



// Custom hook

function useFetchData(url) {

 const [data, setData] = React.useState(null);



 React.useEffect(() ⇒ {
   axios.get(url).then((response) ⇒ setData(response.data));
 }, [url]);

 return data;
}

test('fetches data successfully', async () ⇒ {
 const mockData = { message: 'Hello, world!' };
 mockedAxios.get.mockResolvedValue({ data: mockData });

 const { result, waitForNextUpdate } = renderHook(() ⇒
   useFetchData('https://api.example.com/data')
 );

 await waitForNextUpdate();

 expect(result.current).toEqual(mockData);
});
```

4. Mocking Browser APIs

Hooks like `useEffect` or `useLayoutEffect` that interact with browser APIs (e.g., `localStorage` or `window`) require mocking those APIs.

Example: Testing `localStorage` in a Hook

```
function useLocalStorage(key, initialValue) {
   const [value, setValue] = React.useState(() ⇒ {
     return JSON.parse(localStorage.getItem(key)) || initialValue;
   });
    React.useEffect(() ⇒ {
     localStorage.setItem(key, JSON.stringify(value));
   }, [key, value]);
    return [value, setValue];
 }
  test('stores and retrieves value from localStorage', () ⇒ {
   const key = 'myKey';
   const initialValue = 'default';
    // Mock localStorage
   const mockSetItem = jest.spyOn(Storage.prototype, 'setItem');
   const mockGetItem = jest.spyOn(Storage.prototype, 'getItem');
   mockGetItem.mockReturnValue(JSON.stringify(initialValue));
```

```
    const { result } = renderHook(() ⇒ useLocalStorage(key,
initialValue));

    expect(result.current[0]).toBe(initialValue);

    act(() ⇒ {

     result.current[1]('newValue');

   });

    expect(mockSetItem).toHaveBeenCalledWith(key,
JSON.stringify('newValue'));

 });
```

---

5. Testing React Query or Async Hooks

When testing hooks with asynchronous operations like `React Query` or API fetching, use `@testing-library/react-hooks` or libraries like `msw` to mock the server behavior.

Example: Testing `React Query`

```
import { QueryClient, QueryClientProvider, useQuery } from
'react-query';

import { renderHook } from '@testing-library/react-hooks';

import { act } from 'react-dom/test-utils';

import axios from 'axios';


// Mock axios
```

```
jest.mock('axios');

const mockedAxios = axios as jest.Mocked<typeof axios>;


// Custom hook

function useFetchUser() {

 return useQuery('user', async () ⇒ {

   const { data } = await axios.get('/user');

   return data;

 });

}


test('fetches user data', async () ⇒ {

 const mockData = { name: 'John Doe' };

 mockedAxios.get.mockResolvedValue({ data: mockData });


 const queryClient = new QueryClient();

 const wrapper = ({ children }) ⇒ (

   <QueryClientProvider
client={queryClient}>{children}</QueryClientProvider>

 );
 const { result, waitFor } = renderHook(() ⇒ useFetchUser(), { wrapper
});


 await waitFor(() ⇒ result.current.isSuccess);


 expect(result.current.data).toEqual(mockData);

});
```

Key Takeaways

75. Use `@testing-library/react-hooks` for isolated, direct testing of hooks.
76. Test hooks that involve components using libraries like `@testing-library/react`.
77. Mock APIs, browser APIs, or libraries (like `axios`) as needed for robust tests.
78. Use `act` to handle updates in hooks, ensuring that state changes are processed correctly.

79. What is the React Testing Library, and how does it differ from Enzyme?

React Build and Deployment

76. How do you build a React application for production?

Building a React application for production involves several steps to ensure that the application is optimized, secure, and ready for deployment. Here's a guide to build and prepare a React application for production:

1. Prepare the Application

a. Set Environment Variables

77. Use `.env` files to configure different environments (e.g., development, production).
78. Define `REACT_APP_` prefixed variables for use in the app.

Example `.env.production`:

```
REACT_APP_API_URL=https://api.example.com

REACT_APP_FEATURE_FLAG=true
```

b. Optimize Code

- Remove unused code and components.
- Ensure proper error handling and logging.
- Avoid hardcoding sensitive information (e.g., API keys, secrets).

c. Lint and Test

- Run linters like ESLint to identify issues.
- Execute unit, integration, and end-to-end tests to ensure the app works as expected.

---

2. Build the Application

React's `create-react-app` (CRA) provides a built-in command to generate a production build.

a. Run the Build Command

In the root of your React project, run:

```
npm run build
```

b. What Happens During the Build?

- Minifies JavaScript and CSS to reduce file sizes.
- Optimizes images and other assets.
- Generates a `build` folder with production-ready files:
  - `index.html`
  - Minified JS/CSS files (`static/js/` and `static/css/`).
  - Optimized images (`static/media/`).

---

3. Test the Production Build Locally

Before deploying, test the production build to ensure everything works as expected.

a. Use a Local Server

You can use `serve` to test the `build` directory:

```
npx serve -s build
```

5. Optimize for Production

a. Code Splitting

- Ensure React splits your app into smaller chunks for faster loading.
- Enabled by default in `create-react-app`.

b. Lazy Loading

- Use `React.lazy` and `Suspense` to load components on demand.

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

```
<Suspense fallback={<div>Loading...</div>}>
 <LazyComponent />
</Suspense>
```

c. Caching

- Use cache control headers to cache static files for faster subsequent loads.
- Configure caching in your hosting platform or CDN.

d. Tree Shaking

- Remove unused exports from JavaScript files during the build process.
- Tools like Webpack automatically perform this for you in production builds.

e. Analyze Bundle Size

- Use tools like `source-map-explorer` to analyze the size of your build.

```
npm install source-map-explorer

npx source-map-explorer build/static/js/*.js
```

6. Monitor and Maintain

a. Error Tracking

- Use tools like Sentry or LogRocket to monitor errors in production.

b. Performance Monitoring

- Use tools like Lighthouse or New Relic to monitor performance.

c. Regular Updates

- Keep dependencies updated to fix vulnerabilities and improve performance.
- Use tools like `npm audit` to identify security issues:

```
npm audit fix
```

Example Workflow:

1. Set environment variables in `.env.production`.
2. Run `npm run build`.
3. Test the build locally using `npx serve -s build`.
4. Deploy the `build` folder to your hosting platform.
5. Verify the deployed app and monitor it for errors and performance.

By following these steps, you'll have a React application that's optimized, secure, and ready for a smooth user experience in production.

Yes, you can change the output folder name for the `build` directory in a React project. However, the approach depends on the tools or configurations you use in your React project. Below are some ways to customize the output folder:

For Create React App (CRA)

By default, CRA uses `build` as the output folder. To change it, you'll need to eject the configuration or use a custom script or package.

Option 1: Customize with `react-app-rewired`

Install `react-app-rewired`:

```
npm install react-app-rewired --save-dev
```

1.
2. Create a `config-overrides.js` file in the root of your project.

Add the following code to specify the output directory:

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



module.exports = {
 webpack: (config) ⇒ {
   config.output.path = path.resolve(__dirname, 'custom-folder-name');
// Replace with your desired folder name
   return config;
 },
};
```

3.

Update the `package.json` scripts:

```
 "scripts": {
 "start": "react-app-rewired start",
 "build": "react-app-rewired build",
 "test": "react-app-rewired test"
}
```

4.

Run the build command:

```
npm run build
```

5. The output will now be in the folder you specified (e.g., `custom-folder-name`).

Option 2: Using Webpack Configuration (After Ejecting)

Eject the CRA configuration:

```
npm run eject
```

1. ⚠️ **Warning:** This is irreversible. Proceed only if you're comfortable managing Webpack configurations manually.

2. Open the `webpack.config.js` file in the `config` folder.

Locate the `output` property in the production configuration section and change the `path`:

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



module.exports = {

  // Other configurations...
```

```
 output: {
   path: path.resolve(__dirname, 'custom-folder-name'), // Replace
'custom-folder-name' with your desired folder name
   filename: 'static/js/[name].[contenthash:8].js',
   // other configurations ...
 },
};
```

3.

Run the build command:

```
npm run build
```

4.

For Vite

If you are using Vite instead of CRA, you can change the build folder in `vite.config.js`:

1. Open or create `vite.config.js` in the root of your project.

Add or modify the `build` section:

```
 import { defineConfig } from 'vite';
```

```
export default defineConfig({

 build: {

   outDir: 'custom-folder-name', // Replace with your desired folder
name

 },

});
```

2.

Run the build command:

```
npm run build
```

3.

General Solution: Move the Build Folder

If you're unable or unwilling to modify configurations, you can simply move the build folder after building:

```
npm run build

mv build custom-folder-name
```

While this doesn't change the default output location, it achieves the same result manually.

Best Practices

- **Avoid Ejecting:** Ejecting makes your project harder to maintain. Use tools like `react-app-rewired` or custom scripts where possible.
- **Automation:** If you move the build folder manually, consider writing a post-build script in `package.json` to automate it.

Example:

```
"scripts": {

 "build": "react-scripts build && mv build custom-folder-name"

}
```

This approach ensures minimal effort with consistent results.

79. What is Webpack, and how does it relate to React development?

What is Webpack?

Webpack is a powerful open-source module bundler for JavaScript applications. Its primary purpose is to bundle and manage dependencies for your project by converting your code (JavaScript, CSS, images, etc.) into optimized assets that are ready for production.

Key Features of Webpack

1. **Module Bundling**: Combines multiple files into one or more bundles for efficient loading.
2. **Loaders**: Transforms files (e.g., transpile ES6+ to ES5 using Babel, process CSS/SCSS, or load images).
3. **Plugins**: Enhance functionality (e.g., minification, environment-specific builds).
4. **Code Splitting**: Splits the app into smaller bundles for faster loading.
5. **Tree Shaking**: Removes unused code to optimize the final bundle.
6. **Hot Module Replacement (HMR)**: Updates modules in the browser without a full reload during development.

How Does Webpack Relate to React Development?

1. Module Bundling for React Applications

React applications consist of many modular components written in JavaScript, CSS, and other assets. Webpack bundles these files into a single file (or multiple chunks) for the browser, ensuring efficient delivery.

2. Transpiling Modern JavaScript

React typically uses modern JavaScript (ES6+) and JSX, which browsers don't natively understand. Webpack works with **Babel**, a transpiler, to convert this code into browser-compatible JavaScript.

Example Webpack configuration for Babel:

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/, // Transpile .js files
        exclude: /node_modules/,

        use: {

          loader: 'babel-loader',

          options: {

            presets: ['@babel/preset-env', '@babel/preset-react'],

          },

        },

      },

    ],

  },

};
```

3. Handling Static Assets

Webpack allows you to import and manage assets like CSS, images, and fonts directly in your JavaScript files. This simplifies dependency management in React projects.

Example:

```
import './App.css'; // Handles CSS imports

import logo from './logo.png'; // Handles image imports
```

Webpack uses loaders like `css-loader` and `file-loader` to process these assets.

4. Development Server

Webpack's **development server (Webpack Dev Server)** provides features like live reloading and Hot Module Replacement (HMR), making React development faster and more convenient.

5. Optimizing React Apps for Production

Webpack automatically optimizes React applications for production:

- Minifies JavaScript and CSS.
- Performs tree shaking to remove unused code.
- Splits code into smaller chunks for better performance.

Webpack in Create React App (CRA)

If you're using **Create React App (CRA)**, Webpack is already configured for you:

- Handles JSX and ES6+ with Babel.
- Processes CSS, SCSS, images, and other assets.
- Optimizes the build for production.

You don't need to directly configure Webpack in CRA unless you eject the configuration:

npm run eject

⚠️ **Warning:** Ejecting makes the Webpack configuration accessible for manual changes but harder to manage.

Custom Webpack Configuration for React

If you're not using CRA and want to set up Webpack manually for a React project:

Install Dependencies

npm install webpack webpack-cli webpack-dev-server babel-loader @babel/core @babel/preset-env @babel/preset-react html-webpack-plugin css-loader style-loader --save-dev

Basic `webpack.config.js`

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

const HtmlWebpackPlugin = require('html-webpack-plugin');


module.exports = {

 mode: 'development', // 'production' for optimized builds

 entry: './src/index.js', // Entry file

 output: {

   path: path.resolve(__dirname, 'dist'),

   filename: 'bundle.js', // Output bundle

 },

 module: {

   rules: [

     {

       test: /\.js$/, // Transpile JS and JSX

       exclude: /node_modules/,

       use: 'babel-loader',
     },
     {
       test: /\.css$/, // Process CSS
       use: ['style-loader', 'css-loader'],
     },
     {
       test: /\.(png|jpg|gif|svg)$/, // Load images
       use: 'file-loader',
     },
   ],
 },
 plugins: [
   new HtmlWebpackPlugin({
     template: './public/index.html', // Use HTML template
   }),
 ],
 devServer: {
   static: './dist',
   hot: true, // Enable Hot Module Replacement
 },
};
```

```
Add Scripts to package.json

"scripts": {

 "start": "webpack serve --open",

 "build": "webpack"

}
```

Folder Structure

```
my-react-app/

├── src/

│   ├── index.js  (entry file)

│   ├── App.js    (React component)

│   ├── App.css   (CSS file)

├── public/

│   ├── index.html (HTML template)

├── package.json

├── webpack.config.js
```

Conclusion

Webpack is a crucial tool in React development for bundling, transpiling, optimizing, and managing dependencies. While tools like **Create React App** abstract away most of the complexity, understanding Webpack is valuable for customizing builds, improving performance, and managing complex projects.

80. What is Babel, and why is it used in React applications?

What is Babel?

Babel is a popular JavaScript transpiler that converts modern JavaScript (ES6+), JSX, and TypeScript code into a version of JavaScript that can run in older browsers. It enables developers to write cutting-edge JavaScript features and syntaxes without worrying about browser compatibility.

Why is Babel Used in React Applications?

React applications often use modern JavaScript features and **JSX syntax**, neither of which are natively supported by all browsers. Babel ensures that React applications work seamlessly across different environments by:

1. Transpiling JSX

React uses JSX, a syntax extension that looks similar to HTML but is written inside JavaScript. Browsers cannot interpret JSX directly, so Babel converts it into standard JavaScript.

For example: **JSX Code:**

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

Transpiled Code:

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

---

2. Transpiling Modern JavaScript (ES6+)

React applications often use modern JavaScript features like `let`, `const`, arrow functions, and `async/await`. Babel converts these features into a form compatible with older JavaScript engines.

For example: **Modern JS:**

```
const greet = () => console.log('Hello, World!');
```

Transpiled Code:

```
var greet = function() {

   console.log('Hello, World!');

 };
```

---

3. Supporting Newer Proposals

Babel supports experimental JavaScript features via plugins, allowing developers to use features that are not yet finalized.

Example: Using the optional chaining operator (`?.`):

```
const name = user?.profile?.name;



Babel transpiles this into compatible code:

const name = user && user.profile ? user.profile.name : undefined;
```

Key Features of Babel

1. Plugins

Babel uses plugins to handle specific transformations, such as:

- `@babel/plugin-transform-react-jsx`: Transforms JSX into JavaScript.
- `@babel/plugin-proposal-class-properties`: Supports class property syntax.

2. Presets

Presets are collections of plugins and configurations tailored for specific environments. Popular presets include:

- `@babel/preset-env`: Transpiles modern JavaScript to ES5.
- `@babel/preset-react`: Transpiles JSX and React-specific code.

3. Polyfills

Babel can include polyfills for features not natively supported by certain browsers (e.g., `Promise`, `Map`, `Set`).

How to Use Babel in React Applications

Babel is typically preconfigured in tools like **Create React App** (CRA). However, if you're setting up Babel manually, here's how:

1. Install Babel

Install Babel and necessary presets:

```
npm install @babel/core @babel/cli @babel/preset-env @babel/preset-react
babel-loader --save-dev
```

2. Create a Babel Configuration File

Create a `.babelrc` or `babel.config.js` file in the root of your project:

```
{

   "presets": ["@babel/preset-env", "@babel/preset-react"]

 }
```

3. Integrate Babel with Webpack

Add Babel to your Webpack configuration:

```
module.exports = {

   module: {
     rules: [
       {
         test: /\.js$/,
         exclude: /node_modules/,
         use: {
           loader: "babel-loader",
         },
       },
     ],
   },
 };
```

Babel in Create React App (CRA)

If you're using CRA, Babel is already configured. CRA uses:

- `@babel/preset-env`: To transpile modern JavaScript.
- `@babel/preset-react`: To handle JSX and React-specific code.

You don't need to set up Babel manually unless you eject the CRA configuration.

Benefits of Using Babel in React

1. **Cross-Browser Compatibility**: Ensures your app works in older browsers.

2. **JSX Transformation**: Allows you to use JSX syntax in React.
3. **Future-Proofing**: Enables usage of new JavaScript features before they're widely supported.
4. **Customizability**: Provides plugins and presets to fine-tune transformations.

Summary

Babel is a critical tool in React development that ensures modern JavaScript and JSX code can run on older browsers. It's seamlessly integrated into most React build tools like Create React App but can also be customized for advanced use cases.

These questions require a mix of creativity, deep technical understanding, and practical experience. They help assess a developer's ability to tackle real-world challenges and design scalable, performant React applications.

Here is a list of **unique, scenario-based React.js questions** aimed at assessing the deep skills of an experienced React developer. These questions focus on complex scenarios, optimization strategies, advanced patterns, and real-world applications.

---

1. Performance Optimization in Complex Applications

Scenario: You have a React app that has multiple components re-rendering frequently, especially when the state of a parent component changes. This impacts the performance. How would you optimize the performance of this app?

Answer: Discuss solutions like:

- Using React.memo for memoizing components.
- Leveraging useMemo and useCallback hooks for preventing unnecessary recalculations.
- Splitting large components into smaller ones with lazy loading.
- Utilizing the React Developer Tools to identify unnecessary re-renders.
- Using React.PureComponent or functional components to prevent deep re-renders.

---

2. Handling Large Lists and Virtualization

Scenario: You are building a React app that displays large lists of data (e.g., 10,000+ items). Loading all the items at once is causing performance issues. How would you handle this?

Answer: Discuss the implementation of:

- **Windowing/Virtualization** using libraries like react-window or react-virtualized to render only a subset of the list that is visible in the viewport.
- Lazy loading items on scroll to reduce memory usage.
- Debouncing or throttling events such as search input or list updates.

---

3. Managing Global State with Context API vs Redux

Scenario: You're building a React app with complex state management requirements. You need to decide between using the React Context API or Redux for managing global state. How do you choose which one to use?

Answer: Discuss the trade-offs:

- **Context API:** Good for simpler applications with limited state requirements and fewer updates. Best for small to medium-sized apps.
- **Redux:** More powerful for larger apps with complex state, action handling, middleware (e.g., Redux Thunk), and debugging capabilities.
- Discuss how React Context can cause performance issues with deep component trees due to unnecessary re-renders, while Redux can be more performant with useSelector and memoization techniques.

---

4. Optimizing React App Bundle Size

Scenario: You notice that your React app is growing large in terms of bundle size. How would you reduce the size of the React app's final bundle to improve loading times?

Answer: Strategies to optimize:

- **Code splitting** with React's React.lazy() and Suspense for dynamic imports.
- **Tree shaking** to remove unused code when building the app (enabled by default in Webpack).
- Using lighter libraries and avoiding large dependencies.
- Implementing **asset compression** (e.g., with tools like Webpack's TerserPlugin or using gzip/Brotli).
- **Lazy loading** images, videos, and other assets.
- Using **CDN** for serving third-party libraries.
- Analyzing bundle size with webpack-bundle-analyzer.

---

5. Managing Side Effects with useEffect

Scenario: You need to fetch data when a component mounts and update the UI with the fetched data. However, the fetch call should be made only when the component first mounts, and subsequent renders should not trigger the fetch. How do you handle this with useEffect?

Answer: Use the useEffect hook with an empty dependency array ([]) to fetch data only once when the component mounts:

```
useEffect(() ⇒ {
   fetchData();
 }, []);
```

- Discuss how useEffect works for handling side effects and how to avoid unnecessary fetch calls by properly managing the dependency array.

---

6. Building Custom Hooks

Scenario: You need to build a custom hook that handles the logic for form validation, including error state and form submission handling. How would you structure this custom hook?

Answer: Discuss how to structure a custom hook to manage form state:

- Use useState for form values and errors.
- Use useEffect for handling validation on value changes.
- Return functions and values like handleSubmit, handleChange, and isValid for easy reuse across multiple forms. Example:

```
function useForm(initialValues, validate) {
   const [values, setValues] = useState(initialValues);
   const [errors, setErrors] = useState({});

   const handleChange = (e) ⇒ {
     const { name, value } = e.target;
     setValues((prev) => ({ ...prev, [name]: value }));
     setErrors((prev) => ({ ...prev, [name]: validate[name](value) }));
   };
    const handleSubmit = () => {
     // Submit logic
   };
    return { values, errors, handleChange, handleSubmit };
 }
```

---

7. Implementing Lazy Loading in React

Scenario: You need to implement lazy loading for routes in your React app to improve the performance of the initial load. How would you implement it?

Answer: Use React.lazy() to dynamically import components and wrap them with Suspense:

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

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

- Discuss how to use React.lazy() for route-based code splitting and the performance benefits.
- Mention how Suspense provides a fallback UI during lazy loading.

---

8. Handling Form State Management

Scenario: You're building a complex form with nested fields and dynamic inputs. How would you manage the form state and validation?

Answer: Discuss solutions like:

- Using a controlled form with useState or useReducer for managing complex state.
- Implementing a **form library** like Formik or React Hook Form for easier state management and validation.
- **UseReducer** could be helpful when the state has multiple values and nested structures.
- Example of using useReducer:

```
function formReducer(state, action) {
   switch (action.type) {
     case 'UPDATE_FIELD':
       return { ...state, [action.name]: action.value };
     default:
       return state;
   }
 }
```

---

9. Error Boundaries and Handling UI Crashes

Scenario: Your app has several dynamic components, and you want to ensure that if a JavaScript error occurs in one component, it doesn't crash the entire app. How would you implement this?

Answer: Discuss **Error Boundaries:**

- Create an error boundary component using componentDidCatch or static getDerivedStateFromError to catch errors in components.
- Example:

```
class ErrorBoundary extends React.Component {
   state = { hasError: false };

   static getDerivedStateFromError() {
     return { hasError: true };
   }

   componentDidCatch(error, info) {
     logErrorToMyService(error, info);
   }

   render() {
     if (this.state.hasError) {
       return <h1>Something went wrong.</h1>;
     }

     return this.props.children;
   }
 }
```

---

10. Implementing Debouncing and Throttling

Scenario: You have a search input that triggers API calls on each keystroke. However, you want to reduce the number of API calls and only trigger the API when the user stops typing for a specified period. How would you implement this?

Answer: Use a **debounce** technique:

- Use the useEffect and useState hooks to trigger the search API call after the user stops typing.
- Use a library like **lodash.debounce** for optimized debouncing. Example:

```
const [query, setQuery] = useState('');

const debouncedSearch = useCallback(
 debounce((query) ⇒ fetchSearchResults(query), 500),
 []
);

useEffect(() ⇒ {
 if (query) {
   debouncedSearch(query);
 }
}, [query]);



const handleChange = (e) ⇒ setQuery(e.target.value);
```

---

11. Context API with Dynamic Theme Switching

Scenario: You're building an app where users can toggle between light and dark themes. You need to implement dynamic theme switching using React's Context API.

Answer:

- Create a ThemeContext to store the current theme and a function to toggle it.
- Use useContext in components to access and update the theme.

- Example:

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

function App() {
 const [theme, setTheme] = useState('light');
  const toggleTheme = () => setTheme((prev) => (prev === 'light' ? 
'dark' : 'light'));
  return (
   <ThemeContext.Provider value={{ theme, toggleTheme }}>
     <Navbar />
     <Content />
   </ThemeContext.Provider>
 );
}

function Navbar() {
 const { theme, toggleTheme } = useContext(ThemeContext);
  return (
   <div>
     <h1>{theme === 'light' ? 'Light Mode' : 'Dark Mode'}</h1>
     <button onClick={toggleTheme}>Toggle Theme</button>
   </div>
 );
}
```

---

12. Handling Authentication and Session Management

Scenario: Your app requires user authentication, and you need to persist user login state even after the page reloads. How would you implement this in React?

Answer: Discuss how to handle authentication:

- Use **localStorage** or **sessionStorage** to persist authentication tokens.
- Set the token in HTTP headers using Authorization for

API requests.

- Use **React Context** or **Redux** to store the authenticated user state globally and rehydrate the state on page reload.
- Implement a custom hook for authentication logic like login, logout, and token management.

These questions focus on **real-world challenges** that require deep understanding and problem-solving skills in React.js. They test the candidate's ability to architect solutions for performance, scalability, and maintainability in complex React applications.

Miscellaneous Questions

81. What are portals in React, and when would you use them?
82. What is the difference between server-side rendering (SSR) and client-side rendering (CSR)?
83. What are the main features of React 18?
84. What are Suspense and Concurrent Mode in React?
85. How do you handle animations in React applications?
86. What is the difference between imperative and declarative programming in React?

This list covers a wide range of **basic-level topics** that are essential for understanding ReactJS. It is ideal for evaluating beginners or developers with limited experience in React.

Here's a list of **100 practical ReactJS questions** that could be asked of an experienced developer. These questions cover concepts, coding skills, and debugging scenarios to evaluate hands-on expertise.

React Basics

1. What are React components? How do you create them?
2. What is JSX? How does it work under the hood?
3. Explain the difference between functional and class components.
4. How does React's Virtual DOM differ from the real DOM?
5. What are props? How do you pass and validate them?
6. What is state in React? How do you manage it?
7. How do you update the state in a React component?
8. Can you explain the React lifecycle methods?
9. What are React hooks? List some commonly used hooks.
10. How does the useState hook work? Provide an example.

Advanced React Concepts

11. What is the Context API, and how do you use it?
12. How does React's useEffect hook work? What are its dependencies?
13. What is the difference between useMemo and useCallback hooks?
14. How do you optimize a React application for performance?
15. What is React.memo? How does it help in performance optimization?
16. What is the difference between controlled and uncontrolled components?
17. How do you handle errors in React?
18. Explain the significance of keys in lists.
19. What are higher-order components (HOCs)?
20. What is React's reconciliation process?

State Management

21. How does lifting state up work in React?

22. What are the alternatives to prop drilling in React?
23. How do you use Redux with React? Provide an example.
24. Explain the difference between Redux and the Context API.
25. How do you handle asynchronous actions in Redux?
26. What is Redux Toolkit, and how does it simplify Redux?
27. What are middleware in Redux? Give examples.
28. How would you implement a global state without Redux?
29. How does Zustand or MobX differ from Redux?
30. What is the purpose of the useReducer hook?

Routing

31. What is React Router? How do you set up basic routing?
32. What is the difference between Switch and Routes in React Router v6?
33. How do you handle dynamic routing in React?
34. How do you protect routes in a React application?
35. Explain the useNavigate hook in React Router v6.
36. How would you implement lazy loading with routes?
37. What are nested routes? Provide an example.
38. How do you pass props to a routed component?
39. How do you handle query parameters in React Router?
40. What is the difference between HashRouter and BrowserRouter?

Forms and Events

41. How do you handle forms in React?
42. What is the difference between synthetic events and real DOM events in React?
43. How do you handle input validation in forms?
44. How do you manage form state efficiently?
45. Explain the usage of libraries like Formik or React Hook Form.
46. How do you prevent default behavior in event handlers?
47. How do you debounce input fields in React?
48. How do you manage file uploads in React?
49. What is the purpose of onChange and onSubmit events in forms?
50. How do you handle keyboard events in React?

API Integration

51. How do you fetch data in React? Provide examples.
52. What is the purpose of useEffect in API calls?
53. How would you handle loading, success, and error states in API calls?
54. What is Axios, and how is it different from the Fetch API?
55. How would you cancel an API request in React?
56. How do you handle pagination in API responses?
57. How do you securely store API keys in a React project?
58. What is the difference between client-side and server-side rendering in React?
59. How do you handle authentication with APIs in React?
60. How would you implement infinite scrolling?

Performance Optimization

61. What are the common React performance bottlenecks?
62. How do you use the React DevTools profiler?
63. How does React's batching mechanism work?
64. What is the purpose of the React.PureComponent?
65. How do you implement code splitting in React?
66. What is tree-shaking, and how does it work in React?
67. How do you optimize rendering of large lists in React?
68. How does lazy loading work in React?
69. What is React’s Suspense, and how do you use it?
70. How do you avoid unnecessary re-renders in React?

Testing

71. What is Jest, and how do you use it with React?
72. How do you test React components with Enzyme?
73. How does the React Testing Library differ from Enzyme?
74. How do you mock API calls in tests?
75. How do you test hooks in React?
76. What are snapshots in React testing?
77. How do you test components with props?
78. How do you test form validation?

79. What is the purpose of end-to-end testing in React apps?
80. How do you test event handlers in React?

Build and Deployment

81. How do you configure Webpack for a React project?
82. What is the purpose of Babel in a React application?
83. How do you deploy a React app to production?
84. What is the significance of process.env in React?
85. How do you handle environment-specific configurations in React?
86. What is a service worker, and how do you use it with React?
87. How do you enable HTTPS in a local React development server?
88. What is the purpose of the .env file in React?
89. How do you use Docker to containerize a React application?
90. How would you deploy a React app with CI/CD pipelines?

Debugging

91. How do you debug React applications effectively?
92. How do you handle React-specific error boundaries?
93. How do you debug performance issues in React?
94. What are common issues you encounter while using hooks, and how do you fix them?
95. How do you identify and fix memory leaks in React?
96. How do you handle errors in asynchronous code in React?
97. How do you debug CSS issues in React components?
98. How do you fix "Cannot update a component while rendering a different component" errors?
99. How do you resolve stale closures in hooks?
100. How do you interpret warnings like "React state update on an unmounted component"?

This comprehensive list targets practical skills, problem-solving abilities, and real-world scenarios.

Here is a curated list of **unique, practical ReactJS questions** to assess and deeply probe an experienced developer's skills. These questions are designed to challenge problem-solving, debugging, and advanced understanding of React.

Component Design & Optimization

1. How would you design a React component library that supports theming and custom styles?
2. Explain how you would optimize a component rendering a large data table with frequent updates.
3. How would you implement a reusable and performant debounced search bar component?
4. How do you structure a React project to handle multiple large-scale features efficiently?
5. How would you implement dynamic rendering of different components based on user permissions?

Advanced Hooks Usage

6. How would you recreate the useState hook from scratch?
7. Describe a situation where you would use useLayoutEffect instead of useEffect and why.
8. How would you implement a custom hook for tracking online/offline status across tabs?
9. What are stale closures in React hooks? Provide an example and solution.

10. How do you optimize a useEffect hook that runs on complex dependency arrays?

Context API and State Management

11. Design a context-based global state system without Redux for handling theme and language settings.
12. How would you avoid unnecessary re-renders in deeply nested components using Context API?
13. Implement a pub-sub mechanism using React's Context and hooks.
14. How do you migrate a class-based Redux setup to a hooks-based Redux Toolkit solution?
15. What is the best approach for integrating localStorage with state management in React?

Forms and Validation

16. How would you create a reusable, dynamic form component that handles validation for various field types?
17. How do you optimize forms with hundreds of input fields in React?
18. How would you implement a multi-step form with conditional rendering between steps?
19. Explain how to handle asynchronous validation (e.g., checking username availability) in forms.
20. How would you integrate React Hook Form with third-party UI libraries like Material-UI or Ant Design?

React Performance Tuning

21. How would you debug and fix a slow-rendering React application?
22. Explain how you would implement a memoized pagination component.

23. Describe a situation where React.memo might cause performance issues and how you'd address it.
24. How would you measure and optimize the performance of a React app in production?
25. How would you implement infinite scrolling without affecting app performance?

Error Handling and Resilience

26. How do you design a React app to handle global errors and gracefully recover from them?
27. How would you implement retry logic for failed API requests in React?
28. What is the purpose of error boundaries, and how do you extend them for logging errors to an external service?
29. How do you debug and fix "setState on an unmounted component" warnings?
30. How would you handle React app failures in environments with unreliable internet?

Routing Challenges

31. How would you implement a custom breadcrumb navigation system with React Router?
32. How do you optimize React Router for applications with deeply nested and dynamic routes?
33. How would you implement route-based animations in a React application?
34. Explain how to handle conditional redirects based on user roles in React Router.
35. How do you preload data for a route in a React app before navigating to it?

Integration and Ecosystem

36. How would you integrate server-side rendering (SSR) with React using Next.js or similar frameworks?

37. How would you implement a shared session between a React app and another non-React app?
38. Explain how to securely use third-party libraries in a React app without affecting performance.
39. How would you handle external authentication systems (e.g., OAuth) in a React SPA?
40. How do you integrate WebSockets for real-time updates in a React application?

Debugging Scenarios

41. Debug and resolve a situation where useEffect runs twice in development but not in production.
42. How do you identify memory leaks in React apps and fix them?
43. How would you resolve a problem where a child component is causing a parent component to re-render unnecessarily?
44. How do you debug why a component is rendering multiple times during an interaction?
45. How do you troubleshoot a React app that occasionally freezes or crashes?

React Patterns and Architecture

46. How do you implement the Compound Component pattern in React? Provide an example.
47. What is the Provider pattern, and when would you use it over HOCs?
48. How would you design a plugin system for a React application to allow extensibility?
49. Explain the differences and trade-offs between the Container-Presenter and Render Props patterns.
50. How would you handle feature flags in a React application to toggle features dynamically?

Real-World Problem-Solving

51. How would you migrate a legacy React app to a modern architecture without downtime?
52. How do you design a React app that supports offline mode with automatic syncing?
53. Explain how you would implement dynamic component loading based on user preferences or behavior.
54. How do you manage long-running background tasks in a React app?
55. How do you ensure accessibility compliance (e.g., WCAG standards) in a React project?

Testing

56. How would you test a component with complex hooks and asynchronous effects?
57. How do you mock a Redux store for testing connected components?
58. Explain how you test error boundaries in React.
59. How would you write an end-to-end test for a React app with dynamic routing?
60. How do you test React components that rely on external APIs?

Custom Implementations

61. How would you create a custom drag-and-drop implementation in React?
62. Implement a React component to display a real-time clock with seconds.
63. How would you build a custom tooltip component with delayed hover behavior?
64. How do you implement a dark mode toggle with persisted state across sessions?
65. How would you recreate React's useReducer hook manually?