100 Must-Know React Interview Questions and Answers 2024 – Devinterview.io

devinterview.io/questions/web-and-mobile-development/react-interview-questions/

React Basics

- 1.

  ### What is *React* and why is it used?

  ✓

  Answer:

  React is an open-source, front-end JavaScript library for building user interfaces, that focuses on reusable components and virtual DOM for performance.

  ## Core Concepts

  ### Virtual DOM

  React's Virtual DOM is a lightweight in-memory representation of the actual DOM elements. When changes occur, React compares the current Virtual DOM with a shadow copy and efficiently updates only the changed portions in the real DOM.

  This mechanism significantly reduces expensive direct DOM manipulations, resulting in improved performance and responsiveness in web applications.

  ### Components

  React Components encapsulate both the visual and the corresponding logic. They can be either classes or pure functions. This modular architecture and the ability to nest and reuse components make React a powerful UI toolkit.

  Components are broken down into two main types:

  1. **Class Components:** These are ES6 classes that can hold state and offer lifecycle methods.
  2. **Functional Components:** Primarily plain JavaScript functions: until the advent of "hooks," they couldn't maintain states.

  Key changes, starting from React 16.8:

  - Introduction of new hooks API expanded state-management to functional components
  - Popular hooks include `useState` for state management and `useEffect` for lifecycle management.

  Beyond this foundational structure, hooks offer extensive state, lifecycle, and context APIs, making functional components powerful building blocks.

  ### Unidirectional Data-flow

  React mandates a **one-way** data flow, empowering developers to understand and manage data propagation more effectively. This simplifies tracking, debugging, and validating data changes across the application.

While sibling components can communicate indirectly through shared parent components, direct communication among sibling components is typically discouraged.

JSX: Syntactic Sugar

JSX empowers developers by offering a more intuitive, HTML-like syntax for embedding JavaScript expressions. This marriage of UI and logic not only renders extensive possibilities but also promotes code organization and readability.

Why use React?

Declarative Programming Paradigm

React enables a **declarative style** of programming: developers define the interface's desired state, and React ensures the DOM reflects that state. This approach is more intuitive and helps in designing clear, maintainable code.

Strong Community Backing and Ecosystem

React has been gaining momentum with an enthusiastic community regularly contributing new solutions, updates, and robust third-party libraries. The supportive ecosystem extends to comprehensive toolsets for better development and debugging (like React DevTools).

Reusability and Composability

React's architecture is built on **reusable components**, fostering modular, consistent UI elements and logic that can be redeployed across projects or shared with others.

Performance Optimization

The Virtual DOM serves as a powerful performance amplifier, and features like providing keys to iterated lists ensure efficient and targeted DOM updates. React is also capable of server-side rendering, bolstering app speed and SEO-friendliness.

Effective Data Management

For application-wide state management, React provides **Context API** and libraries like Redux. Meanwhile, local state management with hooks like `useState` streamlines state handling within components.

Code Example: Functional vs Class-Based Components

Here is the React code:

```
// Functional Component with useState hook
import React, { useState } from 'react';

export default function Button() {
  const [count, setCount] = useState(0);
  return (
    <button onClick={() => setCount(count + 1)}>
      Clicked {count} times
    </button>
  );
}

// Class-based Component
import React, { Component } from 'react';

export default class Button extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  render() {
    return (
      <button onClick={() => this.setState({ count: this.state.count + 1 })}>
        Clicked {this.state.count} times
      </button>
    );
  }
}
```

- 2.

How is *React* different from *Angular* or *Vue*?

✓
Answer:

When comparing **React, Angular, and Vue.js**, a few key differentiators stand out.

Core Philosophy

- **React**: Focuses on UI components. You need other libraries for state management, routing, etc.
- **Angular**: Provides a more comprehensive solution out of the box, often called “batteries included.”
- **Vue.js**: A good balance of providing core libraries and flexibility for integration with third-party tools.

Learning Curve

- **React**: Initially simpler to learn due to its focused nature, but can become complex as you add more external libraries.
- **Angular**: Steeper learning curve because of its complete ecosystem, including modules, services, and complex directives.
- **Vue.js**: Known for its gentle learning curve and clear, concise documentation.

Community and Ecosystem

- **React**: Enjoys an enormous community, and users can pick and choose from a vast array of third-party libraries to complement its core features.
- **Angular**: Boasts a comprehensive ecosystem that’s well-managed by its developers and is known for its enterprise support.
- **Vue.js**: While the newest of these frameworks, it has been growing rapidly, with a dedicated team and a flourishing community.

Performance

- **React**: Focuses on efficient rendering and offers built-in tools for performance optimization.
- **Angular**: Optimizes performance through features like Ahead-Of-Time (AOT) compilation and Zone.js, which prevents unnecessary digest cycles.
- **Vue.js**: Also optimized for performance, with a small bundle size and features like lazy-loading components.

Official State Management

- **React**: Employs component state (with `setState`) and also external state management libraries like **Redux**, **MobX**, and the newer **Context API**.
- **Angular**: Primarily uses services and **RxJS** for more structured reactive state management.
- **Vue.js**: Offers Vuex, a state management pattern and library dedicated to Vue applications.

Language Support

- **React**: Developed with **JavaScript** and its supersets (**JSX** and **TypeScript**) in mind.
- **Angular**: Primarily designed for **TypeScript** but supports JavaScript and Dart as well.
- **Vue.js**: Offers support for both **JavaScript** and **TypeScript**.

Templating Approach

- **React**: Utilizes JSX, which combines HTML and JS within JavaScript files. It offers a more concise approach and closely intertwines HTML with JS logic.
- **Angular**: Has a complete separation of concerns with TypeScript, HTML, and CSS in separate files.
- **Vue.js**: Allows for both **single-file components** (SFCs) that encapsulate HTML, JavaScript, and CSS, as well as the traditional trio of separate files.

Language Server Support

- **React**: Known for limited tooling support due to *runtime-oriented* nature, but effective tooling is available for **TypeScript** and **Flow**.
- **Angular**: Offers full **TypeScript** support with features like auto-completion, refactoring, and more, thanks to its built-in language service.
- **Vue.js**: Supports comprehensive programming features, including type verification, integrated debugging, and intelligent code suggestions.

- 3.

  ### What is a *React component*?

  ✓
  Answer:

  A **React component** represents a modular, reusable piece of the user interface. It can encapsulate both **visual elements** (rendered in the Virtual DOM) and **application logic**. React components come in two primary forms: **function components** and **class components**.

  #### Function vs. Class Components

  - **Function Components**: These are stateless, simpler to read, and ideally used for small, specialized UI elements known as ‘dumb’ components. They are pure functions, perceptually faster because of fewer checks.

  - **Class Components**: These can maintain state and expose more advanced features like lifecycle methods. However, the introduction of hooks to function components in React 16.8 technically made state management possible without classes.

  #### JSX and `render()`

  React components generally use JSX (an XML-like syntax) to describe the UI and a `render()` method to define the **visual makeup**.

  - **JSX**: This “syntactic sugar” streamlines component building. It is converted into standard JavaScript calls. Babel is often used to compile this code.
  - **`render()`**: Required for class components, it tells React what the component’s output should be when rendered.

  #### Structural Coherence

  Components in React link together, forming a tree structure. A **root component** is the entry point, and from there, it houses other components.

  #### Data Flow

  React follows a **unidirectional data flow**. This means data moves from the top of the component tree (parent) down to leaves (children) through component **props**. Changes are signaled back up the tree via **callbacks**.

  #### State and Props

  Both function and class components can receive data via two main routes:

  - **Props**: Short for properties, these are akin to function arguments and are immutable. They’re the mechanism for parent-child data transfer.
  - **State**: This is functionally the component’s “memory” and is mutable. Components keep track of their state and re-render upon state change.

Lifecycle Operations

Class components support a series of **lifecycle methods**. These can be used to run code at specific points in the component's lifecycle, such as upon mounting (creation), updating, or unmounting (removal).

Custom classes and the lifecycle methods within were the primary mechanism for side effects earlier in React. While class-based components aren't as central to the framework with the advent of hooks, they're still relevant and in use, especially when using versions < 16.8.1 and realizing the components' lifecycle patterns in codebases.

- 4.

How do you create a *component* in *React*?

✓
Answer:

Creating a React component involves defining its structure, behavior, and sometimes lifecycle methods for dynamic updates. Components can be **functional** or **class-based**.

Code Example

Here's a **Class-based** component:

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

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

And here's a **Functional** one:

```
import React from 'react';

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

Both examples showcase a basic greeting component that takes in a prop `name` and displays a greeting message.

Linters and JSX

Many modern text editors and IDEs support JSX and JavaScript syntax, especially when integrated with linters like ESLint. This setup provides real-time feedback on errors and formatting issues.

Code Styling with AirBNB and Prettier

It's common to see code bases following the **Airbnb** style guide, often coupled with **Prettier** for consistent and automated code formatting.

In the context of component creation, these standards can dictate whether to use single or double quotes for JSX attributes and the method for defining components.

Key Takeaways

- JSX offers a natural, HTML-like syntax for building components in React.
- Components can be function-based or class-based.
- Use modern editing tools and linters for improved code consistency and spotting potential issues in real-time.

- 5.

What is *JSX* and why do we use it in *React*?

✓
Answer:

JSX is a powerful JavaScript Extension that enables the seamless integration of HTML-like structures within React. Notably, it allows for a more **intuitive** component declaration and enhanced developer **productivity**.

Key Features

- **Readable Syntax**: Familiar HTML tags make parsing code and debugging simpler.
- **Component Embedding**: JSX supports direct embedding of components, which enhances modularity.
- **Automatic Babel Conversion**: Behind the scenes, JSX and its HTML-like tags are transpiled into JavaScript for browser compatibility.

Benefits of Using JSX

- **Code Compactness**: JSX helps avoid lengthy `React.createElement` calls.
- **Type Safety**: Modern IDEs provide extensive support for type checking and autocompletion with JSX.
- **Compile-Time Optimizations**: JSX allows for compile-time optimizations, enhancing app performance.
- **Enable Optional Syntax Checks**: For those developing in TypeScript, JSX enables Syntax Checks to ensure code quality.

Code Example: JSX and Its Transpiled Output

Here is the JSX code

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

Here is the equivalent JS code transpiled by Babel:

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

Why Use JSX?

- **Concise Syntax**: JSX provides a succinct, declarative approach to building UIs.

- **Improved Readability**: Its obvious resemblance to HTML promotes code clarity and reduces cognitive load.
- **Static Type Checking**: When used with TypeScript or Flow, JSX brings the benefits of type safety, reducing the probability of runtime errors.
- **Development Efficiency**: By simplifying UI code and providing helpful developer features, JSX accelerates the development process.
- **React Ecosystem Integration**: JSX is the preferred way to write components across the React ecosystem, fostering community best practices.

- 6.

Can you explain the *virtual DOM* in *React*?

✓
Answer:

The **Virtual DOM** is a key concept in React, responsible for its high performance. It efficiently manages the **DOM** setup, minimizes updates, and then syncs them to the actual DOM tree.

How the Virtual DOM Works

1. **Initial Rendering**: When the application starts, React creates a simplified in-memory representation of the DOM, called the Virtual DOM.

2. **Tree Comparison**: During each state change, React builds a new Virtual DOM representation. It then compares this updated representation against the previous one to identify what has changed. This process is often called “reconciliation”.

3. **Selective Rendering**: React determines the most minimal set of changes needed to keep the Virtual DOM in sync with the actual DOM. This approach, known as “reconciliation”, is a performance booster as it reduces unnecessary updates.

4. **Batched Updates**: React performs the actual DOM updates in a batch, typically during the next animation frame or when no more updates are being made. This batching leads to optimized DOM operations, further enhancing performance.

5. **One-Way Sync**: After the in-memory Virtual DOM and the actual DOM have been reconciled and the necessary updates identified, React syncs these changes in a **one-way** process, from the Virtual DOM to the actual DOM. This approach helps prevent unnecessary visual glitches and performance hits.

6. **Asynchronous** Handling: React schedules state changes, ensuring performance by bundling multiple changes that can be processed together. This aids in avoiding unnecessary Virtual DOM updates and ensures efficient tree comparisons.

7. **Preventing Direct DOM Manipulation**: React applications typically avoid manual DOM manipulation. Instead, all changes are made through React, which then uses its Virtual DOM mechanism to batch and apply these changes to the actual DOM.

8. **Support for Cross-Platform Environments**: The Virtual DOM gives sturdy cross-platform capabilities, enabling consistent and optimized performance irrespective of the underlying operating system or hardware.

React’s Virtual DOM is primarily powered through its component architecture and extensive use of JavaScript, fundamentally changing how web applications are built and perform. Its virtuous efficiency is a testament to React’s prowess as a leading front-end framework and contributes to the seamless user experiences React applications are known for providing.

- 7.

What are the differences between a *class component* and a *functional component*?

Answer:

Let’s look at the various aspects and differences between **Class Components** and **Functional Components**.

Core Distinctions

Class Components:

- Utilize the `class` keyword for component definition.
- Can have state management.
- Allow lifecycle methods.
- Are typically verbose.

Functional Components:

- Defined using ES6 functions.
- Lack inherent state or lifecycle management.
- Primarily used for UI representation.
- Introduced `Hooks` in **React 16.8** for state and lifecycle control.

Detail Evaluation

Code Structure

Class Components:

- Consists of a `render()` method.
- Can incorporate other methods for state updates and lifecycle management.

Functional Components:

- Evolved with introduction of React hooks.
- `useState()` and `useEffect()` for state and lifecycle management respectively.

Purpose and Use-Cases

Class Components:

- Suitable for more complex components.
- May be necessary in older codebases.
- Gradually being replaced by hooks and functional components.

Functional Components:

- Focused on UI without managing state.
- Introduced hooks to handle state and lifecycle methods.

Editable State

Class Components:

- Use `this.state` and `this.setState()` to manage state.
- Useful when state contains complex data types.

Functional Components:

- Implement `useState` hook to enable state management in functions.
- Introduced for state management in functional components, simplifying state handling.

Lifecycle Methods

Class Components:

- Offer a wide range of lifecycle methods.
- Example methods include `componentDidMount` and `componentWillUnmount`.

Functional Components:

- Limited lifecycle management before the introduction of hooks.
- Use `useEffect()` to handle actions based on state and props changes.

Context API and Redux Usage

Class Components:

- Can easily be paired with both Context API and Redux.
- Typically used with render props.

Functional Components:

- With hooks like `useContext`, have become proficient in handling shared state.
- Can now be seamlessly integrated with newer global state management libraries like Redux.

Adoption and Transition

- Initial React versions were heavily reliant on class components.
- **Hooks'** introduction in **React 16.8** facilitated the shift towards fully functional components.
- While gradual migration from class to functional is encouraged because of performance benefits, both paradigms can still coexist.

Key Takeaways

- **Class Components**:
  - Traditional class-based components.
  - Prefers `this` context.
  - Houses extensive lifecycle methods.
  - Stands as a more elaborate and structured option.
- **Functional Components**:
  - Evolved to include hooks for state management.
  - Favored for their simplicity and ease of reusability.
  - Perfect for simpler, stateless components.

- 8.

How do you handle *events* in *React*?

✓
Answer:

React simplifies the process of managing and handling events through its use of **synthetic events**.

Handling Events in React

Synthetic Events

React **abstracts browser events** into what are known as *synthetic events*. This ensures a consistent interface across different browsers.

Event Subscription

- When handling events, **React behaves consistently** across all elements, not just form elements.
- React events use *camelCase*, unlike HTML, which is helpful for both **consistency and avoiding reserved words in JavaScript**.
- Use **boolean attributes** in JSX for default browser events.

Special Event Handling

React provides *special interfaces* for certain types of events: input components benefit from the `value` attribute, while media components make use of `src` or other similar attributes specific to their type.

Code Example: Event Handling

Here is the JavaScript code:

```
import React from 'react';

class Form extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: '' };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({ value: event.target.value });
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" value={this.state.value} onChange=
{this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

- 9.

What are *state* and *props* in *React*?

✓
Answer:

In React, **props** and **state** are both used to propagate and manage data. However, they have different roles and management patterns.

Role & Life Cycle

- **Props (short for "properties")** are used **to pass data from a parent component to a child** one. Once passed, props in the child component are read-only and can't be directly modified by the child.

- **State** is used to manage data within a component, and is mutable. Any changes to state values trigger a component re-render.

When to Use

- **Props** are for data that does not change within the component and is provided by a parent.
- **State** is for data that does change within the component and is managed by that component itself.

Management

- When a component receives new props, React will merge them with any existing state. However, it won't override state values unless you explicitly set them.

- Since React re-renders the entire component when you update state, it's important to be efficient in state management. Tools like `useMemo` or `shouldComponentUpdate` can help optimize re-renders.

Unifying with Hooks

- The `useState` hook (along with other hooks like `useEffect`) allows functional components to manage state, bringing them closer in capability to class components.

- Prior to the introduction of hooks in React 16.8, state was the exclusive domain of class components. But now, both state and its associated lifecycle hooks belong to **functional components** as well.

Code Example: State and Props Management

Here is the JavaScript code:

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

// Button Component
const Button = ({ text, color }) => {
  return <button style={{background: color}}>{text}</button>;
};

// ColorPicker Component
const ColorPicker = () => {
  const [color, setColor] = useState('blue');

  const changeColor = (newColor) => {
    setColor(newColor);
  };

  return (
    <div>
      <Button text="Red" color="red" onClick={() => changeColor('red')} />
      <Button text="Blue" color="blue" onClick={() => changeColor('blue')} />
      <Button text="Green" color="green" onClick={() => changeColor('green')}
/>
    </div>
  );
};

// App Component
const App = () => {
  return <ColorPicker />;
};
```

- 10.

  ### How do you pass *data* between *components* in *React*?

  ✓
  Answer:

  Data propagation in React components primarily relies on two mechanisms:

  - **Props**: For unidirectional data flow, parent components pass data to their children via props.

  - **Callback Functions**: Data moves up the tree when children invoke specific functions passed down from their parents.

  Let’s have a look at the best-practices for these two mechanisms.

  #### Using Props

  - **Role**: Primarily used for one-way data flow. The parent furnishes the child with props that the child component can neither alter nor reassign.

  - **Best Practices**:

    - Leverage props for read-only data in child components.
    - Rerender the child component, if necessary, when the prop values change.
  - **Code Example: Read-Only Checkbox**:

  Your task is to write the full code for the React Application to demonstrate passing data to child components using props.

  #### Children Built With Props

  In this code example, `App` maintains the `optionSelected` state that it shares with the `DropDown` and `SelectedOption` components. `DropDown` uses the `optionSelected` state to determine which option was picked, shared with `SelectedOption` to display it.

```
// src/components/DropDown.tsx
interface DropDownProps {
  options: string[]
}

const DropDown: React.FC<DropDownProps> = ({ options }) => {
  const [selected, setSelected] = React.useState(0);

  return (
    <div>
      <div>Options:</div>
      {options.map((opt, index) => (
        <button key={index} onClick={() => setSelected(index)}>{opt}</button>
      ))}
      <SelectedOption option={options[selected]} />
    </div>
  );
};

// src/components/SelectedOption.tsx
interface SelectedOptionProps {
  option: string
}

const SelectedOption: React.FC<SelectedOptionProps> = ({ option }) => {
  return <div>You selected: {option}</div>;
};

// src/App.tsx
const App: React.FC = () => {
  const options = ['Apple', 'Banana', 'Cherry'];
  return <DropDown options={options} />;
};

export default App;
```

React State Management

- 11.

What is a *stateful component*?

✓

Answer:

Stateful components in React are fueled by internal states, allowing them to adapt to user interactions and data changes.

By invoking `this.setState()`, components update their state, triggering a re-render and ensuring the UI and state are in sync.

When to Use

- **Dynamic Interactions**: For components that require dynamic updates, such as a counter that increments on every click.
- **User Input Handling**: Useful for capturing and validating user inputs in forms.
- **Data Fetching**: To manage and display data obtained from API calls.

Code Example: Stateful Component

Here is the JavaScript code:

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

class ClickCounter extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  handleIncrement = () => {
    this.setState(prevState => ({ count: prevState.count + 1 }));
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.handleIncrement}>Increment</button>
      </div>
    );
  }
}

export default ClickCounter;
```

- 12.

Can you explain how *useState* works?

✓
Answer:

`useState` is a built-in **React Hook** that empowers components to preserve stateful values. It amalgamates a stateful value with a state-modifying function, enabling direct manipulation.

Hooks are utility functions that enable you to manage state, side effects, and other React features in **function components**.

Core Components of `useState`

1. **Stateful Value**: The first element in the tuple returned by `useState` carries the current state, like any other state in React.

2. **Setter Function**: The second element is a function that determines the state's new value. Upon invocation, it imparts this new state to the component, just as `setState` does in classes.

Given `value` as the stateful value and `setValue` as the setter function, calling `setValue(newValue)` will alter `value` to `newValue`.

Behavioral Traits of `useState`

- **Lazy Initialization**: If the stateful value necessitates a computationally intensive or time-consuming setup, employing `useState` ensures that this setup occurs exclusively when the component is first rendered rather than on every update.

- **Referential Integrity**: If you employ the `useState` Hook at distinct spots within a component or even dissimilar components, React guarantees that each endeavor manages its unique state underlying value, akin to using `this.state` in classes.

Code Example: useState

Here is the React Component:

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

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

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

- 13.

How do you update the *state* of a *parent component* from a *child component*?

✓
Answer:

React encourages **unidirectional data flow**, primarily passing data from parent to child. However, occasional need arises to update parent state from a child component. This can be facilitated using specific patterns and techniques.

Primary Methods

1. **Props Callback**: Pass a function `onStateChange` as a prop which the child can call to update parent state.

2. **Context API**: Use `Context` to make state accessible and modifiable from descendant components.

Advanced Techniques

1. **UseRef and ForwardRef**: Utilize `useRef` and `forwardRef` to get a reference to a child component, allowing you to directly manipulate its properties.

2. **Global State Management**: Implement a global state management solution like Redux or MobX if state changes are pervasive.

3. **Data Services**: Use a service to manage shared state, which can be updated and read by different components.

Code Example: State Management

Here is the React Component:

```
// App.js - Parent component
import React, { useState } from 'react';
import Child from './Child';

function Parent() {
  const [state, setState] = useState('');

  const updateState = (newState) => {
    setState(newState);
  };

  return <Child updateParentState={updateState} />;
}
```

```
// Child.js - Child component
import React from 'react';

function Child({ updateParentState }) {
  const handleClick = () => {
    updateParentState('New state from child!');
  };

  return <button onClick={handleClick}>Update Parent</button>;
}
```

In this example, the `Parent` component maintains the state, which is updated via the function `updateState` passed as a prop to `Child`. When a button inside `Child` is clicked, the `updateParentState` function updates the parent state.

- 14.

What is *lifting state up* in *React*?

✓

Answer:

Lifting State Up in React entails managing state in parent components to propagate it to multiple children, typically to ensure synchronization or data flow.

Why Use Lifting State Up?

- **Consistent Data**: Prevents inconsistencies in related data scattered across components.
- **Easier Data Modifications**: Minimizes complexity when updating shared data, especially with complex data structures or numerous children.

Core Mechanism: Props

React components communicate using `props`, where child components receive data from parents. During **lifting state up**, the parent maintains the state and passes down relevant data as props.

Lifting State Up in Code

Here is the React code:

Parent Component: RectangleAreaCalculator

```
class RectangleAreaCalculator extends React.Component {
  constructor(props) {
    super(props);
    this.state = { width: 0, height: 0 };
  }

  render() {
    return (
      <div>
        <ShapeInput
          type="number"
          label="Width"
          value={this.state.width}
          onChange={(e) => this.setState({ width: e.target.value })}
        />
        <ShapeInput
          type="number"
          label="Height"
          value={this.state.height}
          onChange={(e) => this.setState({ height: e.target.value })}
        />
        <ShapeArea area={this.state.width * this.state.height} />
      </div>
    );
  }
}
```

Child Components: ShapeInput and ShapeArea

```
const ShapeInput = ({ type, label, value, onChange }) => (
  <div>
    <label>{label}</label>
    <input type={type} value={value} onChange={onChange} />
  </div>
);

const ShapeArea = ({ area }) => <div>Area: {area}</div>;
```

In this example, the `RectangleAreaCalculator` maintains the `width` and `height` state and passes them as props to the `ShapeInput` components. The `ShapeArea` component calculates the area and receives `width` and `height` as props, keeping its state logic-free.

Advantages

- **Single Source of Truth**: Shared data lives in the parent, reducing complexities stemming from data redundancy or inconsistencies.
- **Predictable Data Flow**: Changes to the data layer (parent) trigger updates to all its children. This helps in maintaining the coding standards and data integrity.

Most Common Implementations

- **Form State**: Centralizes form data management in one place, simplifying form submissions or data validation.
- **Shared Logic**: Multiple components using the same data or functionality can benefit from centralized state management.

When It’s Overkill

For small-scale apps or in situations with data that lacks a clear **source of “truth”**, the technique might introduce unnecessary complexity.

Aiming for a balance between centralized and localized state management is key, and React provides tools like `useContext` and `useState` that cater to both requirements.

- 15.

When do you use *Redux* or *Context API* for state management?

Answer:

React Lifecycle & Hooks

- 16.

  Explain the *lifecycle methods* of a *React class component*.

  🔒

  Answer:

- 17.

  How do *hooks* work in *React*?

  🔒

  Answer:

- 18.

  Can you describe the *useEffect hook* and its purpose?

  🔒

  Answer:

- 19.

  How do you *fetch data* with *hooks* in *React*?

  🔒

  Answer:

- 20.

  What rules do you have to follow when using *hooks*?

  🔒

  Answer:

Component Communication

- 21.

  How do *props* work in *React*?

  🔒

  Answer:

- 22.

  What is *prop drilling* and how can you avoid it?

  🔒

  Answer:

- 23.

  Explain the *Context API* and its use cases.

  🔒

  Answer:

- 24.

  How do you use *render props*?

  🔒

  Answer:

- 25.

  What is the *children prop*?

  🔒

  Answer:

Performance Optimization

- 26.

  Why is *performance optimization* important in *React*?

  🔒

  Answer:

- 27.

  What is *React.memo* and when would you use it?

  🔒

  Answer:

- 28.

  How does *PureComponent* differ from *Component* in *React*?

  🔒

  Answer:

- 29.

  Can you explain the concept of *reconciliation* in *React*?

  🔒

  Answer:

- 30.

  How can you prevent unnecessary *re-renders* in *React*?

  🔒

  Answer:

Styling in React

- 31.

  How do you apply *styles* in a *React application*?

  🔒

  Answer:

- 32.

  What is *CSS-in-JS* and how do you implement it in *React*?

  🔒

  Answer:

- 33.

  Can you describe how *Styled-Components* work?

  🔒

  Answer:

- 34.

  What are the advantages of using *Sass* or *LESS* in a *React project*?

  🔒

  Answer:

- 35.

  How do you use *inline styles* in *React*?

  🔒

  Answer:

React Routing

- 36.

  What is *React Router*?

  🔒

  Answer:

- 37.

  How do you create *dynamic routes* in *React*?

  🔒

  Answer:

- 38.

  How would you pass *data* to *routes* in *React Router v5*+?

  🔒

  Answer:

- 39.

  How do you programmatically navigate using *React Router*?

  🔒

  Answer:

- 40.

  What are *route guards* and how can you implement them in *React*?

  🔒

  Answer:

React Patterns

- 41.

  What are *higher-order components (HOCs)*?

  🔒

  Answer:

- 42.

  Explain the *container/presenter* (smart/dumb) **component pattern.**

  🔒

  Answer:

- 43.

  How would you implement a *compound component pattern* in *React*?

  🔒

  Answer:

- 44.

  Explain the use of *custom hooks* in *React*.

  🔒

  Answer:

- 45.

  What is a *render prop pattern*?

  🔒

  Answer:

Form Handling

- 46.

  How do you handle *forms* in *React*?

  🔒

  Answer:

- 47.

  What is *controlled* and *uncontrolled components*?

  🔒

  Answer:

- 48.

  How do you *validate forms* in *React*?

  🔒

  Answer:

- 49.

  What is *Formik* and how is it used in *React forms*?

  🔒

  Answer:

- 50.

  How do you handle *file uploads* in *React*?

  🔒

  Answer:

React with TypeScript

- 51.

  What are the benefits of using *TypeScript* with *React*?

  🔒

  Answer:

- 52.

  How do you define *types* for *props* and *state* in *TypeScript* with *React*?

  🔒

  Answer:

- 53.

  Explain how to use *interfaces* with *React components* and *TypeScript*.

  🔒

  Answer:

- 54.

  How do *TypeScript generics* enhance *react components*?

  🔒

  Answer:

Testing in React

- 55.

  Why is *testing* important in *React*?

  🔒

  Answer:

- 56.

  What are some common *testing libraries* for *React*?

  🔒

  Answer:

- 57.

  How do you test a *React component* with *Jest*?

  🔒

  Answer:

- 58.

  Can you explain the difference between *shallow rendering* and *mount rendering* in *Enzyme*?

  🔒

  Answer:

- 59.

  What is *react-testing-library* and how is it different from *Enzyme*?

  🔒

  Answer:

Advanced React Topics

- 60.

  What are *React fragments* and why are they useful?

  🔒

  Answer:

- 61.

  What is *React portal* and when would you use it?

  🔒

  Answer:

- 62.

  How does *error boundary* work in *React*?

  🔒

  Answer:

- 63.

  What is *server-side rendering* and how is it done with *React*?

  🔒

  Answer:

- 64.

  Can you explain the concept of *suspense* and *lazy loading* in *React*?

  🔒

  Answer:

React and SEO

- 65.

  How does *React* affect *SEO*?

  🔒

  Answer:

- 66.

  What strategies would you use to make a *React application SEO-friendly*?

  🔒

  Answer:

- 67.

  How can *server-side rendering* improve *SEO* with *React applications*?

  🔒

  Answer:

React Native

- 68.

  What is *React Native* and how is it different from *React*?

  🔒

  Answer:

- 69.

  How do you bridge *native modules* in *React Native*?

  🔒

  Answer:

- 70.

  Can you describe the *layout system* in *React Native*?

  🔒

  Answer:

State Management Libraries & GraphQL

- 71.

  What is *Apollo Client* and how does it integrate with *React*?

  🔒

  Answer:

- 72.

  How do you manage *local state* in *Apollo Client*?

  🔒

  Answer:

- 73.

  What is *Redux* and how does it contrast with the *Context API*?

  🔒

  Answer:

- 74.

  Can you detail the *Redux workflow*?

  🔒

  Answer:

- 75.

  How do you handle *side effects* in *Redux applications*?

  🔒

  Answer:

React Development Environment & Tooling

- 76.

  How do you set up a *React project* from scratch?

  🔒

  Answer:

- 77.

  What is *Babel* and why do we use it with *React*?

  🔒

  Answer:

- 78.

  What is *Webpack* and what role does it play in *React development*?

  🔒

  Answer:

- 79.

  How does *hot module replacement* work in *React*?

  🔒

  Answer:

- 80.

  What are the features of *create-react-app* and how do you *eject* from it?

  🔒

  Answer:

Integrations and API Handling

- 81.

  How do you handle *API calls* in *React*?

  🔒

  Answer:

- 82.

  What is *Axios* and how is it used over *fetch* in *React applications*?

  🔒

  Answer:

- 83.

  How would you handle *WebSocket connections* in a *React application*?

  🔒

  Answer:

- 84.

  What are some strategies used to connect a *React front end* to a *backend server*?

  🔒

  Answer:

Deployment and Optimization

- 85.

  How would you deploy a *React application*?

  🔒

  Answer:

- 86.

  How do you optimize the performance of a *React application* for *production*?

  🔒

  Answer:

- 87.

  What are *service workers* and how can they benefit a *React application*?

  🔒

  Answer:

- 88.

  How do you configure *HTTPS* in a *React app*?

  🔒

  Answer:

Accessibility in React

- 89.

  Why is *accessibility* important in *web development*?

  🔒

  Answer:

- 90.

  How can you make a *React application* accessible?

  🔒

  Answer:

- 91.

  What is *ARIA* and how it is used in *React*?

  🔒

  Answer:

Internationalization and Localization

- 92.

  What is *internationalization (i18n)* in *React*?

  🔒

  Answer:

- 93.

  How do you implement *localization (l10n)* in a *React app*?

  🔒

  Answer:

React Code Structure & Best Practices

- 94.

  How do you structure large *React applications*?

  🔒

  Answer:

- 95.

  What are some *best practices* when writing *React code*?

  🔒

  Answer:

- 96.

  How do you ensure *code quality* and *maintainability* in a *React project*?

  🔒

  Answer:

React and Git Workflows

- 97.

  How do you manage *feature branches* in *React development* with *Git*?

  🔒

  Answer:

- 98.

  What are your strategies for resolving *merge conflicts* in *React projects*?

  🔒

  Answer:

React Interviews Problem Solving and Scenarios

- 99.

  How would you handle a *feature request* or *bug report* in an ongoing *React project*?

  🔒

  Answer:

- 100.

  Describe your process for optimizing a *component* that has complex *state logic* and several *child components*.

  🔒

  Answer: