React week 2

22. Component Lifecycle Methods

1. Overview: Component Lifecycle Methods in React are functions that allow you to hook into different stages of a component's life. These stages include mounting, updating, and unmounting. Understanding these methods is crucial for performing actions at specific points in a component’s lifecycle, such as fetching data, setting up subscriptions, or cleaning up resources.

2. Example: Consider a class-based component that fetches data from an API when it is first rendered and cleans up a timer when the component is removed from the DOM.

3. Code Example:

javascript
Copy code

```
class DataFetcher extends React.Component {
  constructor(props) {
    super(props);
    this.state = { data: null };
  }

  componentDidMount() {
    // Fetch data when the component is mounted
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => this.setState({ data }));
  }

  componentWillUnmount() {
    // Clean up when the component is unmounted
    clearInterval(this.timerID);
  }

  render() {
    return (
      <div>
        <h1>Fetched Data</h1>
        <pre>{JSON.stringify(this.state.data, null, 2)}</pre>
      </div>
```

```
    );
  }
}
```

4. Explanation:

- **componentDidMount:** This method is invoked immediately after a component is mounted (inserted into the tree). It's a good place to initiate network requests or set up any subscriptions.
- **componentWillUnmount:** This method is called just before a component is unmounted and destroyed. It's the right place to clean up anything that won't automatically be cleaned up, such as timers or subscriptions.

5. Summary: Component Lifecycle Methods provide hooks to run code at specific times during a component's existence. componentDidMount is typically used for data fetching, and componentWillUnmount for cleanup tasks.

https://drive.google.com/file/d/1kFTW7B_kE_tEH7LB5kH2AhPLeaJr4ITY/preview

---

23. Component Mounting Lifecycle Methods

1. Overview: Component Mounting Lifecycle Methods are invoked during the process of putting a component into the DOM. These methods include constructor, getDerivedStateFromProps, render, and componentDidMount.

2. Example: A simple component that sets its state based on props during mounting.

3. Code Example:

javascript
Copy code

```
class Welcome extends React.Component {
  constructor(props) {
    super(props);
    this.state = { greeting: '' };
  }

  static getDerivedStateFromProps(nextProps, prevState) {
    if (nextProps.name !== prevState.name) {
      return { greeting: `Hello, ${nextProps.name}!` };
```

```
  }
  return null;
 }

 componentDidMount() {
  console.log('Component has mounted!');
 }

 render() {
  return <h1>{this.state.greeting}</h1>;
 }
}
```

4. Explanation:

- **constructor:** Used for initializing state and binding methods.
- **getDerivedStateFromProps:** Allows you to update the state based on props. It's a static method that is rarely needed but can be useful in specific cases.
- **render:** The only required method in a class component, responsible for rendering the UI.
- **componentDidMount:** Runs after the component has been rendered to the DOM, making it ideal for operations that require DOM nodes.

5. Summary: Mounting lifecycle methods handle the creation and insertion of a component into the DOM. They are crucial for setting up initial state and making DOM-related operations.

https://drive.google.com/file/d/1QqOmopejZkvAjwdVc2gYnVBrWT23z5_u/preview

---

24. Component Updating Lifecycle Methods

1. Overview: Component Updating Lifecycle Methods are called when a component is being re-rendered due to changes in props or state. These methods include `getDerivedStateFromProps`, `shouldComponentUpdate`, `render`, `getSnapshotBeforeUpdate`, and `componentDidUpdate`.

2. Example: A component that decides whether to update based on specific state changes.

3. Code Example:

javascript
Copy code

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

  shouldComponentUpdate(nextProps, nextState) {
    // Only update if the count is an even number
    return nextState.count % 2 === 0;
  }

  componentDidUpdate(prevProps, prevState) {
    console.log('Component did update!');
  }

  render() {
    return (
      <div>
        <h1>{this.state.count}</h1>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Increment
        </button>
      </div>
    );
  }
}
```

4. Explanation:

- **shouldComponentUpdate:** Determines whether the component should re-render based on state or props changes.
- **componentDidUpdate:** Invoked after the component has been updated. It's a good place to operate on the DOM when the component has been re-rendered.

5. Summary: Updating lifecycle methods manage re-renders and allow developers to control when and how a component should update, optimizing performance and managing side effects.

https://drive.google.com/file/d/1cjWLjUAjMwQYG2Ibjis-ZBbqRW8FgWth/preview

25. Fragments

1. Overview: Fragments in React allow you to group a list of children without adding extra nodes to the DOM. This is useful when you want to return multiple elements from a component's render method without wrapping them in an additional DOM element.

2. Example: A component returning multiple elements without a wrapper `div`.

3. Code Example:

javascript
Copy code

```
function List() {
  return (
    <React.Fragment>
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
    </React.Fragment>
  );
}
```

4. Explanation:

- **React.Fragment:** A wrapper that does not render any additional DOM element. It can be used to avoid unnecessary `div` elements that might mess up your CSS or create unnecessary nesting in the DOM.

5. Summary: Fragments provide a way to group multiple elements without introducing extra nodes into the DOM, keeping the DOM cleaner and more manageable.

https://drive.google.com/file/d/15JMl4BRzbkm8RHR4qZ_hTjpjgXzQOTPL/preview

26. memo

1. Overview: `React.memo` is a higher-order component that optimizes functional components by preventing unnecessary re-renders. It only re-renders the component if its props have changed.

2. Example: A functional component that only re-renders when its props change.

3. Code Example:

javascript
Copy code
```
const ExpensiveComponent = React.memo(function({ data }) {
  console.log('Rendering ExpensiveComponent');
  return <div>{data}</div>;
});
```

4. Explanation:

- **React.memo:** This function takes a component and returns a memoized version of it. If the component's props remain the same, React skips the rendering process, thus improving performance.

5. Summary: React.memo helps optimize functional components by skipping unnecessary re-renders, especially useful for components that require complex calculations or have heavy rendering logic.

https://drive.google.com/file/d/1WG_GFWeBbdRaAcxN2Y6EY9Bj3U-dr5JH/preview

---

27. Refs

1. Overview: Refs provide a way to access DOM nodes or React elements created in the render method. They are commonly used for managing focus, text selection, triggering animations, or integrating with third-party DOM libraries.

2. Example: A component that focuses an input field when it mounts.

3. Code Example:

javascript
Copy code
```
class InputFocus extends React.Component {
  constructor(props) {
    super(props);
    this.inputRef = React.createRef();
  }
```

```
  componentDidMount() {
    this.inputRef.current.focus();
  }

  render() {
    return <input type="text" ref={this.inputRef} />;
  }
}
```

4. Explanation:

- **React.createRef:** Used to create a ref, which can be attached to any React element via the ref attribute.
- **this.inputRef.current:** Gives you direct access to the DOM node or element associated with the ref.

5. Summary: Refs provide a powerful way to interact directly with DOM elements in React, allowing you to control them imperatively in a way that complements React's declarative nature.

https://drive.google.com/file/d/18N0JMQ5ORzhmvCT0YggV9sDGpZXBljZp/preview

—----------

28. Refs with Class Components

1. Overview: Refs in class components are used to directly access and manipulate DOM elements or React elements created in the render method. They allow you to interact with these elements outside the typical React data flow, which is particularly useful for tasks like managing focus, media playback, or triggering animations.

2. Example: A class component that programmatically clicks a button when it mounts.

3. Code Example:

javascript
Copy code

```
class ButtonClicker extends React.Component {
  constructor(props) {
    super(props);
```

```
    this.buttonRef = React.createRef();
  }

  componentDidMount() {
    this.buttonRef.current.click();
  }

  render() {
    return (
      <button ref={this.buttonRef}>
        Click me!
      </button>
    );
  }
}
```

4. Explanation:

- **React.createRef:** Used to create a reference to a DOM element in the component.
- **this.buttonRef.current:** Accesses the underlying DOM element. This allows you to call methods directly on it, such as click, focus, or other DOM operations.

5. Summary: Using refs in class components is a powerful way to interact directly with DOM elements, giving you control over imperative tasks that might be difficult to achieve with React's declarative approach.

https://drive.google.com/file/d/1ZqxmmAHyjqq17ngePPWG0G-ecA2Q--Mb/preview

---

29. Forwarding Refs

1. Overview: Forwarding refs is a technique for passing a ref through a component to one of its child components. This is useful when you want to expose a child component's DOM node or component instance to a parent component.

2. Example: A button component that forwards its ref to an inner DOM element.

3. Code Example:

javascript
Copy code

```
const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="fancy-button">
    {props.children}
  </button>
));

class App extends React.Component {
  constructor(props) {
    super(props);
    this.buttonRef = React.createRef();
  }

  componentDidMount() {
    this.buttonRef.current.focus();
  }

  render() {
    return <FancyButton ref={this.buttonRef}>Click Me</FancyButton>;
  }
}
```

4. Explanation:

- **React.forwardRef:** This function creates a component that forwards its ref to a child component. The forwarded ref can then be used as if it was defined directly in the parent component.
- **ref forwarding:** Allows a component to be more flexible by exposing its internal DOM elements to parent components, which can then directly interact with them.

5. Summary: Forwarding refs makes it easier to work with components that encapsulate other components or DOM nodes, allowing you to pass down refs transparently and control child elements from a parent component.

https://drive.google.com/file/d/1LSkJ-HeEJE0SJt8VQDkobOYMBFe54K5g/preview

30. Portals

1. Overview: Portals in React provide a way to render children into a DOM node that exists outside the DOM hierarchy of the parent component. This is particularly useful for modals, tooltips, or any UI element that needs to break out of its parent container due to styling or positioning constraints.

2. Example: A modal component rendered outside the parent DOM hierarchy.

3. Code Example:

javascript
Copy code

```
class Modal extends React.Component {
  render() {
    return ReactDOM.createPortal(
      <div className="modal">
        {this.props.children}
      </div>,
      document.getElementById('modal-root')
    );
  }
}

// Usage
function App() {
  return (
    <div>
      <h1>Main App</h1>
      <Modal>
        <p>This is a modal!</p>
      </Modal>
    </div>
  );
}
```

4. Explanation:

- **ReactDOM.createPortal:** This method creates a portal, which allows you to render a component's children into a different part of the DOM, specified by the second argument.

- **Portals:** Useful for UI components that require different positioning or styling from the rest of the application, such as modals or popups.

5. Summary: Portals provide a way to render components outside their usual parent DOM hierarchy, which is useful for creating flexible and reusable UI patterns that require specific DOM placement.

https://drive.google.com/file/d/1LSkJ-HeEJE0SJt8VQDkobOYMBFe54K5g/preview

---

31. Error Boundary

1. Overview: Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of crashing the whole application. They help improve the robustness and user experience of your app by preventing the entire UI from breaking due to an error in a single component.

2. Example: An error boundary component that catches errors in its child components.

3. Code Example:

javascript
Copy code

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    console.error("Error caught:", error, info);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
```

```
    }

    return this.props.children;
  }
}

// Usage
function BuggyComponent() {
  throw new Error("I crashed!");
}

function App() {
  return (
    <ErrorBoundary>
      <BuggyComponent />
    </ErrorBoundary>
  );
}
```

4. Explanation:

- **getDerivedStateFromError:** This lifecycle method is used to update the state to indicate an error was caught.
- **componentDidCatch:** Logs the error information, which can be useful for debugging or sending to a monitoring service.
- **Error boundaries:** Can be used to wrap parts of your application to catch errors without breaking the entire UI, allowing for graceful degradation.

5. Summary: Error boundaries are essential for handling exceptions in a React app, ensuring that errors in one part of the UI don't crash the entire application. They enhance stability and user experience by providing fallback UIs.

https://drive.google.com/file/d/1YUHVqghKvjVm6qUqYEhiRLbk81k8tHnQ/preview

---

32. Higher Order Components (Part 1)

1. Overview: Higher Order Components (HOCs) are a pattern in React that allows you to reuse component logic. An HOC is a function that takes a component and returns a new component with additional props or behavior. They are often used to add cross-cutting concerns, such as logging, caching, or conditional rendering, without modifying the original component.

2. Example: An HOC that adds logging functionality to a wrapped component.

3. Code Example:

javascript
Copy code

```
function withLogging(WrappedComponent) {
  return class extends React.Component {
    componentDidMount() {
      console.log(`${WrappedComponent.name} mounted`);
    }

    render() {
      return <WrappedComponent {...this.props} />;
    }
  };
}

// Usage
class MyComponent extends React.Component {
  render() {
    return <div>Hello, World!</div>;
  }
}

const MyComponentWithLogging = withLogging(MyComponent);
```

4. Explanation:

- **HOCs:** Allow you to create a wrapper component that adds additional behavior or props to the wrapped component. This pattern is powerful for reusing code and abstracting common functionality.
- **withLogging:** In this example, the HOC adds logging functionality, logging a message whenever the wrapped component mounts.

5. Summary: Higher Order Components are a powerful React pattern that enables code reuse by wrapping components with additional logic. They are particularly useful for adding cross-cutting concerns to components in a clean and reusable manner.

https://drive.google.com/file/d/1-zjVBu7C-kAE1K7zAPjXBDnZRV5stAw5/preview

—----------

33. Higher Order Components (Part 2)

1. Overview: In this part, we'll delve deeper into Higher Order Components (HOCs) by exploring how to handle props, state, and lifecycle methods within HOCs. We'll also discuss the potential pitfalls of using HOCs, such as "wrapping hell" and how to mitigate them.

2. Example: An HOC that enhances a component by injecting additional props and managing state.

3. Code Example:

javascript
Copy code

```
function withEnhancements(WrappedComponent) {
 return class extends React.Component {
  constructor(props) {
   super(props);
   this.state = { count: 0 };
   this.increment = this.increment.bind(this);
  }

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

  render() {
   return (
    <WrappedComponent
     {...this.props}
     count={this.state.count}
     increment={this.increment}
    />
```

```
    );
   }
  };
}

// Usage
function Counter({ count, increment }) {
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}

const EnhancedCounter = withEnhancements(Counter);
```

4. Explanation:

- **State Management in HOCs:** The HOC manages its state and passes it down to the wrapped component, allowing for complex logic to be abstracted away from the original component.
- **Prop Injection:** The HOC can also inject additional props or functions, like `increment`, which can then be used by the wrapped component.

5. Summary: HOCs can manage state and inject additional props into components, making them powerful tools for creating reusable and enhanced components. However, they must be used carefully to avoid overly complex component hierarchies.

https://drive.google.com/file/d/1Y9sE9lybqeM-Bxv2qZvz4nfethexszQI/preview

---

34. Higher Order Components (Part 3)

1. Overview: In the final part of HOCs, we'll explore real-world use cases and best practices, including how to chain multiple HOCs and the benefits of using HOCs for cross-cutting concerns like authentication, logging, and caching.

2. Example: Chaining multiple HOCs to add multiple layers of functionality to a component.

3. Code Example:

javascript
Copy code

```
function withLogging(WrappedComponent) {
  return class extends React.Component {
    componentDidMount() {
      console.log(`${WrappedComponent.name} mounted`);
    }

    render() {
      return <WrappedComponent {...this.props} />;
    }
  };
}

function withAuthentication(WrappedComponent) {
  return class extends React.Component {
    render() {
      const isAuthenticated = true; // Simulate authentication
      if (!isAuthenticated) {
        return <div>Please log in</div>;
      }
      return <WrappedComponent {...this.props} />;
    }
  };
}

// Usage
class MyComponent extends React.Component {
  render() {
    return <div>Protected Content</div>;
  }
}

const ProtectedComponent = withLogging(withAuthentication(MyComponent));
```

4. Explanation:

- **Chaining HOCs:** Multiple HOCs can be combined to add various layers of functionality. In this example, `withAuthentication` checks if the user is authenticated, and `withLogging` logs when the component mounts.
- **Best Practices:** When chaining HOCs, ensure that each layer adds distinct, reusable functionality to keep the codebase maintainable.

5. Summary: HOCs can be chained together to add multiple layers of functionality, making them ideal for handling cross-cutting concerns. Following best practices helps keep HOCs manageable and maintainable in larger projects.

https://drive.google.com/file/d/1klp4f0M53l6GU4t-UNfgqsmr4dtHefhn/preview

---

35. Render Props (Part 1)

1. Overview: Render Props is a pattern in React that involves passing a function as a prop to a component, allowing you to share logic between components without using HOCs. This approach provides more flexibility and control over how components are rendered.

2. Example: A component that tracks mouse movement and uses a render prop to display the coordinates.

3. Code Example:

javascript
Copy code

```
class MouseTracker extends React.Component {
  constructor(props) {
    super(props);
    this.state = { x: 0, y: 0 };
    this.handleMouseMove = this.handleMouseMove.bind(this);
  }

  handleMouseMove(event) {
    this.setState({ x: event.clientX, y: event.clientY });
  }

  render() {
```

```
    return (
      <div onMouseMove={this.handleMouseMove}>
        {this.props.render(this.state)}
      </div>
    );
  }
}

// Usage
function App() {
  return (
    <MouseTracker
      render={({ x, y }) => (
        <p>The mouse position is ({x}, {y})</p>
      )}
    />
  );
}
```

4. Explanation:

- **Render Props:** This pattern involves passing a function (the render prop) to a component, which it uses to determine what to render based on the state or props passed to it. This allows you to share and reuse logic between components.
- **Flexibility:** Render Props provide more flexibility than HOCs as they allow the child component to have more control over the rendering logic.

5. Summary: Render Props offer a flexible way to share logic between components, providing an alternative to HOCs that allows for more control over the rendering process.

https://drive.google.com/file/d/1qULHOz1gcLHcEiNwNpyel3OkTymn7dOu/preview

---

36. Render Props (Part 2)

1. Overview: In this part, we'll explore more advanced usage of Render Props, including how to manage multiple render props and best practices for avoiding unnecessary re-renders.

2. Example: A component that handles both mouse and window resizing events using render props.

3. Code Example:

javascript
Copy code

```
class WindowResizer extends React.Component {
  constructor(props) {
    super(props);
    this.state = { width: window.innerWidth, height: window.innerHeight };
    this.handleResize = this.handleResize.bind(this);
  }

  handleResize() {
    this.setState({
      width: window.innerWidth,
      height: window.innerHeight
    });
  }

  componentDidMount() {
    window.addEventListener('resize', this.handleResize);
  }

  componentWillUnmount() {
    window.removeEventListener('resize', this.handleResize);
  }

  render() {
    return this.props.render(this.state);
  }
}

// Usage
function App() {
  return (
    <WindowResizer
      render={({ width, height }) => (
        <p>Window size: {width}x{height}</p>
      )}
    />
```

```
  );
}
```

4. Explanation:

- **Managing Multiple Render Props:** You can manage multiple render props by passing different state or props to the render functions, allowing for complex logic to be shared across components.
- **Avoiding Re-renders:** To avoid unnecessary re-renders, consider using React.memo or other optimization techniques when using Render Props, as the function passed can change on each render.

5. Summary: Advanced usage of Render Props allows for sharing more complex logic between components. Proper management and optimization can prevent performance issues and ensure that components remain responsive.

https://drive.google.com/file/d/1LMpwqeSKJJS0IAEL8eSDPgDsw--Fi-Fj/preview

---

37. Context (Part 1)

1. Overview: The Context API in React allows you to share values (like theme, user information, or locale) across components without explicitly passing props through every level of the tree. This is especially useful for global data that many components need to access.

2. Example: A theme context that provides the current theme to all components in the tree.

3. Code Example:

javascript
Copy code

```
const ThemeContext = React.createContext('light');

class ThemeProvider extends React.Component {
  constructor(props) {
    super(props);
    this.state = { theme: 'light' };
    this.toggleTheme = this.toggleTheme.bind(this);
  }
```

```
  toggleTheme() {
    this.setState((prevState) => ({
      theme: prevState.theme === 'light' ? 'dark' : 'light'
    }));
  }

  render() {
    return (
      <ThemeContext.Provider value={this.state.theme}>
        {this.props.children}
        <button onClick={this.toggleTheme}>Toggle Theme</button>
      </ThemeContext.Provider>
    );
  }
}

function ThemedComponent() {
  return (
    <ThemeContext.Consumer>
      {(theme) => <div className={`theme-${theme}`}>Current theme: {theme}</div>}
    </ThemeContext.Consumer>
  );
}

// Usage
function App() {
  return (
    <ThemeProvider>
      <ThemedComponent />
    </ThemeProvider>
  );
}
```

4. Explanation:

- **Context API:** Provides a way to share data across the component tree without passing props down manually at every level.
- **Provider and Consumer:** The Provider component provides the context value to its descendants, and the Consumer component or the useContext hook can be used to access that value.

5. Summary: The Context API simplifies the management of global data that needs to be accessed by many components. It reduces the need for prop drilling and makes the codebase cleaner and easier to manage

https://drive.google.com/file/d/1JM_Y8XLcnyK8dqplAPMyEZEaRTNQ_ona/preview

—----------

38. Context (Part 2)

1. Overview: In this part, we'll explore how to update the context dynamically and how to handle multiple contexts in a React application. This is important for managing more complex state scenarios where different parts of your application rely on different pieces of shared data.

2. Example: A language context that can be updated dynamically, alongside a theme context.

3. Code Example:

javascript
Copy code

```
const ThemeContext = React.createContext('light');
const LanguageContext = React.createContext('en');

class AppProvider extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      theme: 'light',
      language: 'en',
      toggleTheme: this.toggleTheme,
      switchLanguage: this.switchLanguage
    };
  }

  toggleTheme = () => {
    this.setState((prevState) => ({
      theme: prevState.theme === 'light' ? 'dark' : 'light'
    }));
  };

  switchLanguage = (lang) => {
    this.setState({ language: lang });
```

```
  };

  render() {
    return (
      <ThemeContext.Provider value={this.state.theme}>
        <LanguageContext.Provider value={this.state.language}>
          {this.props.children}
          <button onClick={() => this.toggleTheme()}>Toggle Theme</button>
          <button onClick={() => this.switchLanguage('fr')}>Switch to French</button>
        </LanguageContext.Provider>
      </ThemeContext.Provider>
    );
  }
}

function ThemedLanguageComponent() {
  return (
    <ThemeContext.Consumer>
      {(theme) => (
        <LanguageContext.Consumer>
          {(language) => (
            <div className={`theme-${theme}`}>
              Current theme: {theme}, Language: {language}
            </div>
          )}
        </LanguageContext.Consumer>
      )}
    </ThemeContext.Consumer>
  );
}

// Usage
function App() {
  return (
    <AppProvider>
      <ThemedLanguageComponent />
    </AppProvider>
  );
}
```

4. Explanation:

- **Dynamic Context Updates:** You can pass functions through context that allow child components to update the context value dynamically, like toggling the theme or switching the language.
- **Handling Multiple Contexts:** React allows you to nest multiple Provider components, making it easy to manage different pieces of global state in different contexts.

5. Summary: Dynamically updating context and managing multiple contexts are advanced features that allow for flexible and scalable state management in React applications. They are particularly useful in large-scale applications with diverse global states.

https://drive.google.com/file/d/1HIjcBN23QuOoC7qLNqcUdAFbNYb7a1X-/preview

---

39. Context (Part 3)

1. Overview: In this final part, we'll discuss best practices for using the Context API in large applications, including performance considerations, avoiding unnecessary re-renders, and structuring your context to keep your codebase maintainable.

2. Example: A performance-optimized context setup using React.memo and custom hooks.

3. Code Example:

javascript
Copy code

```
const ThemeContext = React.createContext();
const LanguageContext = React.createContext();

const useTheme = () => React.useContext(ThemeContext);
const useLanguage = () => React.useContext(LanguageContext);

const ThemeProvider = React.memo(({ children }) => {
  const [theme, setTheme] = React.useState('light');

  const toggleTheme = () => {
    setTheme((prevTheme) => prevTheme === 'light' ? 'dark' : 'light');
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
```

```
      {children}
    </ThemeContext.Provider>
  );
});

const LanguageProvider = React.memo(({ children }) => {
  const [language, setLanguage] = React.useState('en');

  const switchLanguage = (lang) => {
    setLanguage(lang);
  };

  return (
    <LanguageContext.Provider value={{ language, switchLanguage }}>
      {children}
    </LanguageContext.Provider>
  );
});

function App() {
  return (
    <ThemeProvider>
      <LanguageProvider>
        <ThemedLanguageComponent />
      </LanguageProvider>
    </ThemeProvider>
  );
}

function ThemedLanguageComponent() {
  const { theme, toggleTheme } = useTheme();
  const { language, switchLanguage } = useLanguage();

  return (
    <div className={`theme-${theme}`}>
      <p>Current theme: {theme}</p>
      <p>Language: {language}</p>
      <button onClick={toggleTheme}>Toggle Theme</button>
      <button onClick={() => switchLanguage('fr')}>Switch to French</button>
    </div>
  );
```

```
}
```

4. Explanation:

- **Performance Optimization:** By using React.memo, you can prevent unnecessary re-renders of the context providers, ensuring that only the components that need to update are re-rendered.
- **Custom Hooks:** Custom hooks like useTheme and useLanguage make context usage more modular and reusable across different parts of your application.

5. Summary: Optimizing context for performance and structuring it effectively are crucial for maintaining a responsive and scalable React application. Following best practices ensures that your context usage remains efficient and your codebase stays clean.

https://drive.google.com/file/d/11RDjsy9-CXw3-JLhPLtVCCMvKGY8D9ij/preview

---

40. HTTP and React

1. Overview: React is often used to build applications that interact with backend services via HTTP requests. Understanding how to make HTTP requests and handle responses in React is essential for building dynamic, data-driven applications.

2. Example: A simple React component that fetches data from an API when it mounts.

3. Code Example:

javascript
Copy code

```
class DataFetcher extends React.Component {
  constructor(props) {
    super(props);
    this.state = { data: null, loading: true };
  }

  componentDidMount() {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => this.setState({ data, loading: false }))
      .catch(error => console.error('Error fetching data:', error));
  }
```

```
 render() {
   const { data, loading } = this.state;

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

   return (
     <div>
       <h1>Data from API:</h1>
       <pre>{JSON.stringify(data, null, 2)}</pre>
     </div>
   );
  }
}

// Usage
function App() {
  return <DataFetcher />;
}
```

4. Explanation:

- **HTTP Requests:** In React, you can use the fetch API to make HTTP requests to backend services. Handling the response involves converting it to JSON and updating the component state.
- **Component Lifecycle:** The data fetch is initiated in the componentDidMount lifecycle method, ensuring the request is made after the component is mounted.

5. Summary: Understanding how to perform HTTP requests in React is fundamental for building applications that interact with backend services. It allows you to create dynamic, data-driven user interfaces.

https://drive.google.com/file/d/1H5PcvLBDWtKjg_ZjoU6-GFJUFeuC6HyU/preview

---

41. HTTP GET Request

1. Overview: The HTTP GET method is used to retrieve data from a server at a specified resource. In React, GET requests are commonly used to fetch data when a component mounts or when a specific user action triggers data retrieval.

2. Example: A React component that performs a GET request to display a list of users.

3. Code Example:

javascript
Copy code

```
class UserList extends React.Component {
  constructor(props) {
    super(props);
    this.state = { users: [], loading: true };
  }

  componentDidMount() {
    fetch('https://api.example.com/users')
      .then(response => response.json())
      .then(users => this.setState({ users, loading: false }))
      .catch(error => console.error('Error fetching users:', error));
  }

  render() {
    const { users, loading } = this.state;

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

    return (
      <ul>
        {users.map(user => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
    );
  }
}

// Usage
function App() {
```

```
  return <UserList />;
}
```

4. Explanation:

- **GET Requests:** The GET method is used to fetch data from the server. In this example, the component fetches a list of users from an API and displays them.
- **Component State:** The data retrieved from the GET request is stored in the component's state, and the UI is updated accordingly once the data is loaded.

5. Summary: HTTP GET requests are essential for retrieving data from servers in React applications. They are commonly used to populate components with data when they are first rendered or in response to user actions.

https://drive.google.com/file/d/1APmdWIq70PlOlwq8RZQRuYFjeX7App4_/preview

---

42. HTTP POST Request

1. Overview: The HTTP POST method is used to send data to a server, typically to create or update a resource. In React, POST requests are often used in forms to submit user input to a backend service.

2. Example: A React component that submits a form using a POST request.

3. Code Example:

javascript
Copy code

```
class UserForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: '', email: '' };
  }

  handleSubmit = (event) => {
    event.preventDefault();

    fetch('https://api.example.com/users', {
```

```
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(this.state)
  })
    .then(response => response.json())
    .then(data => console.log('User created:', data))
    .catch(error => console.error('Error creating user:', error));
 };

 handleChange = (event) => {
  this.setState({ [event.target.name]: event.target.value });
 };

 render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <input
        type="text"
        name="name"
        value={this.state.name}
        onChange={this.handleChange}
        placeholder="Name"
      />
      <input
        type="email"
        name="email"
        value={this.state.email}
        onChange={this.handleChange}
        placeholder="Email"
      />
      <button type="submit">Submit</button>
    </form>
   );
  }
}

// Usage
function App() {
 return <UserForm />;
}
```

4. Explanation:

- **POST Requests:** The POST method is used to send data to the server, often to create new records. This example shows a simple form submission using a POST request.
- **Form Handling:** The form's state is managed in React, and the handleSubmit method handles the POST request when the form is submitted.

5. Summary: HTTP POST requests are crucial for sending data to servers in React applications. They are commonly used in forms to submit user input, allowing for interaction with backend services and database updates.

https://drive.google.com/file/d/1474GU3Ol2nYxFqJhM_j_wKnTvuJJqB_1/preview

43. React Hooks Tutorial-1-Introduction

1. Overview: In this introductory tutorial, we'll explore what React Hooks are, why they were introduced, and how they revolutionize the way React components are written. Hooks allow you to use state and other React features without writing a class, making function components more powerful and flexible.

2. Example: A comparison between a class component and a functional component using Hooks to manage state.

3. Code Example:

javascript

Copy code

```
// Class Component with State

class CounterClass extends React.Component {

  constructor(props) {

    super(props);

    this.state = { count: 0 };

  }
```

```
  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };

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

// Functional Component with useState Hook
function CounterHook() {
  const [count, setCount] = React.useState(0);

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

  return (
```

```
    <div>
      <p>Hook Component Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

4. Explanation:

- **Why Hooks:** Hooks were introduced to eliminate the need for class components in scenarios where state or lifecycle methods were required, simplifying the code and making it easier to understand and maintain.
- **Comparison:** The example shows how a functional component using Hooks can achieve the same functionality as a class component but with less code and better readability.

5. Summary: React Hooks provide a new, simpler way to manage state and side effects in functional components. They allow you to reuse logic across components without changing your component hierarchy, making your React code more modular and maintainable.

https://drive.google.com/file/d/1GfVvI8QY-FRfvx0zKnbwtD1KgFemr0LS/preview

---

React Hooks Tutorial - 2 - useState Hook

1. Overview: The `useState` hook is the most basic and commonly used hook in React. It allows you to add state to functional components. This tutorial will cover how to use `useState` to manage and update the state in a React component.

2. Example: A functional component that uses `useState` to toggle between two states.

3. Code Example:

javascript

Copy code

```
function ToggleSwitch() {
  const [isOn, setIsOn] = React.useState(false);


  const toggle = () => {
    setIsOn(!isOn);
  };


  return (
    <div>
      <p>The switch is {isOn ? 'ON' : 'OFF'}</p>
      <button onClick={toggle}>Toggle</button>
    </div>
  );
}
```

4. Explanation:

- **useState Syntax:** useState returns an array with two elements: the current state value and a function to update that state. You can call this function to update the state, which will trigger a re-render of the component.
- **Initial State:** The initial state is passed as an argument to useState. In this example, false is the initial state of the isOn variable.

5. Summary: The useState hook is fundamental in React for managing state in functional components. It enables you to store values and update them, triggering re-renders to reflect changes in your UI.

https://drive.google.com/file/d/1xVQ0GMaa6vz3_WDDodFLdMB5tl7-MIR7/preview

---

React Hooks Tutorial - 3 - useState with Previous State

1. Overview: This tutorial will delve deeper into the useState hook by showing how to update state based on the previous state value. This is crucial when the new state depends on the old state, such as incrementing a counter or toggling a value.

2. Example: A counter component that increments the count based on the previous count value.

3. Code Example:

javascript

Copy code

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

  const increment = () => {
    setCount(prevCount => prevCount + 1);
  };

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

```
}
```

4. Explanation:

- **Updating Based on Previous State:** When updating state based on the previous state, it's recommended to use the callback form of the state updater function, which takes the previous state as an argument and returns the new state.
- **Why This Matters:** This approach ensures that your state updates correctly even if multiple updates are queued, which can happen in asynchronous scenarios.

5. Summary: Using the useState hook with the previous state is essential for scenarios where the new state depends on the old one. It ensures that state updates are consistent and accurate, especially in complex or asynchronous environments.

https://drive.google.com/file/d/1dmGDC6rE9kVIwS1w0DSwyF9jP6nQEfI8/preview

MCQ

Topic: Component Lifecycle Methods

Question 1:
Which of the following is NOT a phase in the React component lifecycle?
A) Initialization
B) Mounting
C) Destruction
D) Updating

Answer: C) Destruction

Topic: Component Mounting Lifecycle Methods

Question 2:
Which of the following lifecycle methods is invoked right after a component is added to the DOM?
A) componentWillUnmount()
B) componentDidMount()
C) shouldComponentUpdate()
D) render()

Answer: B) componentDidMount()

Topic: Component Updating Lifecycle Methods

Question 3:
Which method is used to prevent a component from re-rendering in the update phase?
A) shouldComponentUpdate()
B) componentDidUpdate()
C) getDerivedStateFromProps()
D) render()

Answer: A) shouldComponentUpdate()

Topic: Fragments

Question 4:
What is the primary use of React Fragments (<React.Fragment></React.Fragment>)?
A) To conditionally render JSX
B) To return multiple elements without adding extra nodes to the DOM
C) To catch JavaScript errors in child components
D) To improve performance in lists

Answer: B) To return multiple elements without adding extra nodes to the DOM

Topic: memo

Question 5:
The React memo function is used to:

A) Log component errors
B) Optimize functional components by preventing unnecessary re-renders
C) Manage side effects in a component
D) Automatically bind `this` to class methods

Answer: B) Optimize functional components by preventing unnecessary re-renders

---

Topic: Refs

Question 6:
What is the purpose of Refs in React?
A) To manage state
B) To handle side effects
C) To directly access or modify DOM elements
D) To pass props between components

Answer: C) To directly access or modify DOM elements

Topic: Refs with Class Components

Question 7:
In class components, how do you create a ref?
A) `this.createRef()`
B) `React.useRef()`
C) `React.createRef()`
D) `useEffect()`

Answer: C) `React.createRef()`

Topic: Forwarding Refs

Question 8:
What is the purpose of React's ref forwarding?
A) To pass refs to other functional components
B) To optimize component rendering

C) To avoid unnecessary re-renders
D) To catch errors in child components

Answer: A) To pass refs to other functional components

Topic: Portals

Question 9:
What is the main purpose of React Portals?
A) To optimize state management
B) To render components outside of the parent DOM hierarchy
C) To handle errors in component trees
D) To optimize functional components

Answer: B) To render components outside of the parent DOM hierarchy

---

Topic: Error Boundary

Question 10:
Which lifecycle method is commonly used in React Error Boundaries to catch errors in child components?
A) `getDerivedStateFromError()`
B) `componentDidCatch()`
C) `componentWillUnmount()`
D) `render()`

Answer: B) `componentDidCatch()`