**Exercise 1: Implement componentDidMount**

Create a React component that fetches data from an API when the component is first mounted and displays the data in a list format.

**Solution**

import React, { Component } from "react";

class DataFetcher extends Component {

state = {

data: [],

loading: true,

};

componentDidMount() {

// Simulate API call

setTimeout(() => {

this.setState({

data: ["Item 1", "Item 2", "Item 3"],

loading: false,

});

}, 2000);

}

render() {

const { data, loading } = this.state;

return (

<div>

<h1>Data List</h1>

{loading ? <p>Loading...</p> : <ul>{data.map((item, index) => <li key={index}>{item}</li>)}</ul>}

</div>

);

}

}

export default DataFetcher;

**Exercise 2: Use componentDidUpdate**

Create a counter component that logs a message to the console whenever the count value is updated.

**Solution**

import React, { Component } from "react";

class Counter extends Component {

state = {

count: 0,

};

increment = () => {

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

};

componentDidUpdate(prevProps, prevState) {

if (prevState.count !== this.state.count) {

console.log(`Count updated: ${this.state.count}`);

}

}

render() {

return (

<div>

<h1>Counter</h1>

<p>Count: {this.state.count}</p>

<button onClick={this.increment}>Increment</button>

</div>

);

}

}

export default Counter;

**Exercise 3: Cleanup with componentWillUnmount**

Create a component that sets up an interval to update a timestamp every second and clears the interval when the component is unmounted.

**Solution**

import React, { Component } from "react";

class Timer extends Component {

state = {

time: new Date().toLocaleTimeString(),

};

componentDidMount() {

this.interval = setInterval(() => {

this.setState({ time: new Date().toLocaleTimeString() });

}, 1000);

}

componentWillUnmount() {

clearInterval(this.interval);

console.log("Timer component unmounted, interval cleared.");

}

render() {

return (

<div>

<h1>Current Time</h1>

<p>{this.state.time}</p>

</div>

);

}

}

export default Timer;

**Exercise 4: Use getDerivedStateFromProps**

Create a component that updates its state based on changes in its props.

**Solution**

import React, { Component } from "react";

class DerivedState extends Component {

state = {

message: this.props.initialMessage,

};

static getDerivedStateFromProps(nextProps, prevState) {

if (nextProps.initialMessage !== prevState.message) {

return { message: nextProps.initialMessage };

}

return null;

}

render() {

return (

<div>

<h1>Message: {this.state.message}</h1>

</div>

);

}

}

export default DerivedState;

**Exercise 5: Use shouldComponentUpdate**

Create a component that re-renders only if the value of the text prop changes.

**Solution**

import React, { Component } from "react";

class ConditionalRender extends Component {

shouldComponentUpdate(nextProps) {

return nextProps.text !== this.props.text;

}

render() {

console.log("Rendered");

return <div>{this.props.text}</div>;

}

}

export default ConditionalRender;

**Exercise 6: Advanced Exercise: Combine Lifecycle Methods**

Create a component that:

1. Fetches data when mounted.
2. Logs updates whenever the props or state change.
3. Cleans up any subscriptions when unmounted.

**Solution**

import React, { Component } from "react";

class CombinedLifecycle extends Component {

state = {

data: [],

loading: true,

};

componentDidMount() {

console.log("Component mounted.");

setTimeout(() => {

this.setState({ data: ["Item 1", "Item 2", "Item 3"], loading: false });

}, 2000);

}

componentDidUpdate(prevProps, prevState) {

console.log("Component updated.");

if (prevState.data !== this.state.data) {

console.log("Data updated:", this.state.data);

}

}

componentWillUnmount() {

console.log("Component will unmount.");

}

render() {

const { data, loading } = this.state;

return (

<div>

<h1>Combined Lifecycle Methods</h1>

{loading ? <p>Loading...</p> : <ul>{data.map((item, index) => <li key={index}>{item}</li>)}</ul>}

</div>

);

}

}

export default CombinedLifecycle;

**Exercise 7: Fetch Data with useEffect (Equivalent to componentDidMount)**

Create a functional component that fetches data when it mounts using the useEffect hook.

**Solution**

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

const DataFetcherWithHooks = () => {

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

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

useEffect(() => {

// Simulate API call

setTimeout(() => {

setData(["Item 1", "Item 2", "Item 3"]);

setLoading(false);

}, 2000);

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

return (

<div>

<h1>Data List</h1>

{loading ? <p>Loading...</p> : <ul>{data.map((item, index) => <li key={index}>{item}</li>)}</ul>}

</div>

);

};

export default DataFetcherWithHooks;

**Exercise 8: Cleanup with useEffect (Equivalent to componentWillUnmount)**

Create a functional component that sets up an interval to update a timestamp and cleans it up when the component is unmounted.

**Solution**

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

const TimerWithHooks = () => {

const [time, setTime] = useState(new Date().toLocaleTimeString());

useEffect(() => {

const interval = setInterval(() => {

setTime(new Date().toLocaleTimeString());

}, 1000);

// Cleanup function

return () => {

clearInterval(interval);

console.log("Timer component unmounted, interval cleared.");

};

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

return (

<div>

<h1>Current Time</h1>

<p>{time}</p>

</div>

);

};

export default TimerWithHooks;

**Exercise 9: Conditional Re-Rendering with Hooks**

Create a functional component that only updates the title of the page (document.title) when a counter is incremented.

**Solution**

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

const CounterWithHooks = () => {

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

useEffect(() => {

document.title = `Count: ${count}`;

console.log(`Count updated to ${count}`);

}, [count]); // Runs only when `count` changes

return (

<div>

<h1>Counter</h1>

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

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

</div>

);

};

export default CounterWithHooks;

**Exercise 10: Simulating shouldComponentUpdate with React.memo**

Create a functional component that prevents unnecessary re-renders using React.memo.

**Solution**

import React, { useState } from "react";

const ConditionalRenderWithHooks = React.memo(({ text }) => {

console.log("Rendered");

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

});

const ParentComponent = () => {

const [text, setText] = useState("Hello");

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

return (

<div>

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

<button onClick={() => setText("Updated Text")}>Update Text</button>

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

<ConditionalRenderWithHooks text={text} />

</div>

);

};

export default ParentComponent;

**Exercise 11: Combining Lifecycle Methods with Hooks**

Create a functional component that:

1. Fetches data on mount.
2. Logs updates to the state.
3. Cleans up subscriptions when unmounted.

**Solution**

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

const CombinedHooks = () => {

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

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

useEffect(() => {

console.log("Component mounted.");

const fetchData = () => {

setTimeout(() => {

setData(["Item 1", "Item 2", "Item 3"]);

setLoading(false);

}, 2000);

};

fetchData();

return () => {

console.log("Component unmounted.");

};

}, []); // Runs on mount and unmount

useEffect(() => {

if (!loading) {

console.log("Data updated:", data);

}

}, [data]); // Runs when `data` changes

return (

<div>

<h1>Combined Lifecycle with Hooks</h1>

{loading ? <p>Loading...</p> : <ul>{data.map((item, index) => <li key={index}>{item}</li>)}</ul>}

</div>

);

};

export default CombinedHooks;

**Exercise 12: Implementing getSnapshotBeforeUpdate**

Create a component that displays a chat box. When new messages are added, the component scrolls to the bottom to display the latest message.

**Solution**

import React, { Component } from "react";

class ChatBox extends Component {

constructor(props) {

super(props);

this.chatRef = React.createRef();

this.state = {

messages: ["Welcome to the chat!"],

};

}

addMessage = () => {

this.setState((prevState) => ({

messages: [...prevState.messages, `New message ${prevState.messages.length + 1}`],

}));

};

getSnapshotBeforeUpdate(prevProps, prevState) {

if (prevState.messages.length < this.state.messages.length) {

return this.chatRef.current.scrollHeight;

}

return null;

}

componentDidUpdate(prevProps, prevState, snapshot) {

if (snapshot !== null) {

this.chatRef.current.scrollTop = snapshot;

}

}

render() {

return (

<div>

<h1>Chat Box</h1>

<div

ref={this.chatRef}

style={{

border: "1px solid black",

height: "200px",

overflowY: "scroll",

padding: "10px",

}}

>

{this.state.messages.map((msg, index) => (

<p key={index}>{msg}</p>

))}

</div>

<button onClick={this.addMessage}>Add Message</button>

</div>

);

}

}

export default ChatBox;

**Exercise 13: Using componentDidCatch for Error Boundaries**

Create a parent component that catches errors in a child component and displays a fallback UI.

**Solution**

import React, { Component } from "react";

class ErrorBoundary extends Component {

state = { hasError: false };

componentDidCatch(error, info) {

console.error("Error caught in ErrorBoundary:", error, info);

this.setState({ hasError: true });

}

render() {

if (this.state.hasError) {

return <h1>Something went wrong.</h1>;

}

return this.props.children;

}

}

class ErrorProneComponent extends Component {

state = { throwError: false };

render() {

if (this.state.throwError) {

throw new Error("An intentional error!");

}

return (

<div>

<h1>Error Prone Component</h1>

<button onClick={() => this.setState({ throwError: true })}>Trigger Error</button>

</div>

);

}

}

const App = () => (

<ErrorBoundary>

<ErrorProneComponent />

</ErrorBoundary>

);

export default App;

**Exercise 14: Prevent Re-Renders with shouldComponentUpdate**

Create a component that displays user data but only re-renders when the user ID changes.

**Solution**

import React, { Component } from "react";

class UserData extends Component {

shouldComponentUpdate(nextProps) {

return nextProps.userId !== this.props.userId;

}

render() {

console.log("Rendered UserData component");

return <div>User ID: {this.props.userId}</div>;

}

}

class App extends Component {

state = { userId: 1 };

changeUser = () => {

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

};

render() {

return (

<div>

<h1>User Data</h1>

<UserData userId={this.state.userId} />

<button onClick={this.changeUser}>Change User</button>

</div>

);

}

}

export default App;

**Exercise 15: Using componentWillUnmount for Cleanup**

Create a countdown timer component that clears its interval when the component is unmounted.

**Solution**

import React, { Component } from "react";

class Countdown extends Component {

state = { timeLeft: 10 };

componentDidMount() {

this.timer = setInterval(() => {

this.setState((prevState) => ({

timeLeft: prevState.timeLeft - 1,

}));

}, 1000);

}

componentWillUnmount() {

clearInterval(this.timer);

console.log("Countdown component unmounted.");

}

render() {

return (

<div>

<h1>Countdown: {this.state.timeLeft}s</h1>

</div>

);

}

}

class App extends Component {

state = { showCountdown: true };

toggleCountdown = () => {

this.setState((prevState) => ({

showCountdown: !prevState.showCountdown,

}));

};

render() {

return (

<div>

<button onClick={this.toggleCountdown}>

{this.state.showCountdown ? "Stop Countdown" : "Start Countdown"}

</button>

{this.state.showCountdown && <Countdown />}

</div>

);

}

}

export default App;

**Exercise 16: Fetch Data with componentDidMount and Update UI**

Create a component that fetches a list of posts from a simulated API and displays them. Add a button to refresh the list.

**Solution**

import React, { Component } from "react";

class PostList extends Component {

state = {

posts: [],

loading: true,

};

fetchPosts = () => {

this.setState({ loading: true });

setTimeout(() => {

this.setState({

posts: [

"Post 1: React Lifecycle Methods",

"Post 2: Component State Management",

"Post 3: Handling Events in React",

],

loading: false,

});

}, 2000);

};

componentDidMount() {

this.fetchPosts();

}

render() {

const { posts, loading } = this.state;

return (

<div>

<h1>Posts</h1>

{loading ? (

<p>Loading...</p>

) : (

<ul>

{posts.map((post, index) => (

<li key={index}>{post}</li>

))}

</ul>

)}

<button onClick={this.fetchPosts}>Refresh Posts</button>

</div>

);

}

}

export default PostList;

**Exercise 17: Conditionally Fetch Data with componentDidUpdate**

Create a component that fetches data based on a selected category and updates the display when the category changes. Ensure the component does not fetch data unnecessarily.

**Solution**

import React, { Component } from "react";

class CategoryFetcher extends Component {

state = {

category: "Technology",

articles: [],

};

fetchArticles = (category) => {

console.log(`Fetching articles for category: ${category}`);

setTimeout(() => {

this.setState({

articles: [

`${category} Article 1`,

`${category} Article 2`,

`${category} Article 3`,

],

});

}, 1000);

};

componentDidMount() {

this.fetchArticles(this.state.category);

}

componentDidUpdate(prevProps, prevState) {

if (prevState.category !== this.state.category) {

this.fetchArticles(this.state.category);

}

}

changeCategory = (newCategory) => {

this.setState({ category: newCategory });

};

render() {

return (

<div>

<h1>Category: {this.state.category}</h1>

<div>

<button onClick={() => this.changeCategory("Technology")}>

Technology

</button>

<button onClick={() => this.changeCategory("Health")}>Health</button>

<button onClick={() => this.changeCategory("Finance")}>Finance</button>

</div>

<h2>Articles:</h2>

<ul>

{this.state.articles.map((article, index) => (

<li key={index}>{article}</li>

))}

</ul>

</div>

);

}

}

export default CategoryFetcher;

**Exercise 18: Timer with componentWillUnmount for Cleanup**

Create a stopwatch component that starts counting when mounted and stops counting when unmounted. The parent component should control the mounting and unmounting of the stopwatch.

**Solution**

import React, { Component } from "react";

class Stopwatch extends Component {

state = { seconds: 0 };

componentDidMount() {

this.timer = setInterval(() => {

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

}, 1000);

}

componentWillUnmount() {

clearInterval(this.timer);

}

render() {

return <h1>Time: {this.state.seconds}s</h1>;

}

}

class App extends Component {

state = { showStopwatch: true };

toggleStopwatch = () => {

this.setState((prevState) => ({ showStopwatch: !prevState.showStopwatch }));

};

render() {

return (

<div>

<button onClick={this.toggleStopwatch}>

{this.state.showStopwatch ? "Stop Stopwatch" : "Start Stopwatch"}

</button>

{this.state.showStopwatch && <Stopwatch />}

</div>

);

}

}

export default App;

**Exercise 19: Dynamic Theme Changer**

Create a component that changes its background color when the theme (dark or light) is toggled. Use componentDidUpdate to log theme changes to the console.

**Solution**

import React, { Component } from "react";

class ThemeChanger extends Component {

state = {

theme: "light",

};

toggleTheme = () => {

this.setState((prevState) => ({

theme: prevState.theme === "light" ? "dark" : "light",

}));

};

componentDidUpdate(prevProps, prevState) {

if (prevState.theme !== this.state.theme) {

console.log(`Theme changed to: ${this.state.theme}`);

}

}

render() {

const { theme } = this.state;

const style = {

backgroundColor: theme === "light" ? "#fff" : "#333",

color: theme === "light" ? "#000" : "#fff",

padding: "20px",

textAlign: "center",

};

return (

<div style={style}>

<h1>Current Theme: {theme}</h1>

<button onClick={this.toggleTheme}>

Switch to {theme === "light" ? "Dark" : "Light"} Theme

</button>

</div>

);

}

}

export default ThemeChanger;

**Exercise 20: Lazy Loading with componentDidMount**

Create a component that simulates lazy-loading a large dataset, showing a "Loading..." message until the data is loaded.

**Solution**

import React, { Component } from "react";

class LazyLoader extends Component {

state = {

data: null,

};

componentDidMount() {

setTimeout(() => {

this.setState({

data: Array.from({ length: 50 }, (\_, index) => `Item ${index + 1}`),

});

}, 2000);

}

render() {

const { data } = this.state;

return (

<div>

<h1>Lazy Loader</h1>

{data ? (

<ul>

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

<li key={index}>{item}</li>

))}

</ul>

) : (

<p>Loading...</p>

)}

</div>

);

}

}

export default LazyLoader;

**Exercise 21: shouldComponentUpdate Optimization**

Create a counter component that only re-renders when the count is divisible by 5.

**Solution**

import React, { Component } from "react";

class OptimizedCounter extends Component {

state = { count: 0 };

increment = () => {

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

};

shouldComponentUpdate(nextProps, nextState) {

return nextState.count % 5 === 0;

}

render() {

console.log("Rendering Counter...");

return (

<div>

<h1>Count: {this.state.count}</h1>

<button onClick={this.increment}>Increment</button>

</div>

);

}

}

export default OptimizedCounter;

**Exercise 22: Chat Application Simulation**

Create a chat application component that simulates receiving new messages every 5 seconds. Use componentDidMount to start the simulation and componentWillUnmount to stop it.

**Solution**

import React, { Component } from "react";

class ChatApp extends Component {

state = {

messages: [],

isActive: true,

};

componentDidMount() {

this.simulateMessages();

}

componentWillUnmount() {

clearInterval(this.messageInterval);

}

simulateMessages = () => {

this.messageInterval = setInterval(() => {

this.setState((prevState) => ({

messages: [

...prevState.messages,

`New message at ${new Date().toLocaleTimeString()}`,

],

}));

}, 5000);

};

toggleChat = () => {

const { isActive } = this.state;

if (isActive) {

clearInterval(this.messageInterval);

} else {

this.simulateMessages();

}

this.setState({ isActive: !isActive });

};

render() {

const { messages, isActive } = this.state;

return (

<div>

<h1>Chat Application</h1>

<button onClick={this.toggleChat}>

{isActive ? "Stop Chat" : "Resume Chat"}

</button>

<ul>

{messages.map((msg, index) => (

<li key={index}>{msg}</li>

))}

</ul>

</div>

);

}

}

export default ChatApp;

**Exercise 23: Dynamic Form Builder**

Create a component that builds a dynamic form based on a configuration object. Use componentDidUpdate to log changes when the form configuration updates.

**Solution**

import React, { Component } from "react";

class DynamicForm extends Component {

state = {

formConfig: {

fields: [

{ label: "Name", type: "text" },

{ label: "Email", type: "email" },

{ label: "Age", type: "number" },

],

},

formData: {},

};

handleChange = (field, value) => {

this.setState((prevState) => ({

formData: { ...prevState.formData, [field]: value },

}));

};

componentDidUpdate(prevProps, prevState) {

if (prevState.formConfig !== this.state.formConfig) {

console.log("Form configuration updated");

}

}

updateFormConfig = () => {

this.setState({

formConfig: {

fields: [

...this.state.formConfig.fields,

{ label: "Phone", type: "text" },

],

},

});

};

render() {

const { formConfig, formData } = this.state;

return (

<div>

<h1>Dynamic Form</h1>

<form>

{formConfig.fields.map((field, index) => (

<div key={index}>

<label>{field.label}</label>

<input

type={field.type}

value={formData[field.label] || ""}

onChange={(e) =>

this.handleChange(field.label, e.target.value)

}

/>

</div>

))}

</form>

<button onClick={this.updateFormConfig}>Add Phone Field</button>

<pre>{JSON.stringify(formData, null, 2)}</pre>

</div>

);

}

}

export default DynamicForm;

**Exercise 24: Data Synchronization with External API**

Simulate a component that fetches and synchronizes data with an external API every 10 seconds. Use componentDidMount to start the sync process and componentWillUnmount to stop it.

**Solution**

import React, { Component } from "react";

class DataSync extends Component {

state = {

data: [],

lastSync: null,

};

componentDidMount() {

this.fetchData();

this.syncInterval = setInterval(this.fetchData, 10000);

}

componentWillUnmount() {

clearInterval(this.syncInterval);

}

fetchData = () => {

console.log("Fetching data...");

setTimeout(() => {

const timestamp = new Date().toLocaleTimeString();

this.setState({

data: [`Data item at ${timestamp}`],

lastSync: timestamp,

});

}, 2000);

};

render() {

const { data, lastSync } = this.state;

return (

<div>

<h1>Data Synchronization</h1>

<p>Last Sync: {lastSync || "Never"}</p>

<ul>

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

<li key={index}>{item}</li>

))}

</ul>

</div>

);

}

}

export default DataSync;

**Exercise 25: Scroll Position Saver**

Create a component that saves the scroll position of the page and restores it when the component is remounted. Use componentDidMount and componentWillUnmount for this.

**Solution**

import React, { Component } from "react";

class ScrollSaver extends Component {

componentDidMount() {

window.scrollTo(0, sessionStorage.getItem("scrollPosition") || 0);

window.addEventListener("scroll", this.saveScrollPosition);

}

componentWillUnmount() {

window.removeEventListener("scroll", this.saveScrollPosition);

}

saveScrollPosition = () => {

sessionStorage.setItem("scrollPosition", window.scrollY);

};

render() {

return (

<div style={{ height: "2000px", padding: "10px" }}>

<h1>Scroll Saver</h1>

<p>Scroll and refresh the page to see the effect.</p>

</div>

);

}

}

export default ScrollSaver;

**Exercise 26: E-Commerce Cart Tracker**

Build a shopping cart component that tracks items added or removed. Log changes to the cart whenever the state is updated using componentDidUpdate.

**Solution**

import React, { Component } from "react";

class CartTracker extends Component {

state = {

cart: [],

};

addItem = (item) => {

this.setState((prevState) => ({

cart: [...prevState.cart, item],

}));

};

removeItem = (item) => {

this.setState((prevState) => ({

cart: prevState.cart.filter((i) => i !== item),

}));

};

componentDidUpdate(prevProps, prevState) {

if (prevState.cart !== this.state.cart) {

console.log("Cart updated:", this.state.cart);

}

}

render() {

const { cart } = this.state;

return (

<div>

<h1>Shopping Cart</h1>

<button onClick={() => this.addItem("Item 1")}>Add Item 1</button>

<button onClick={() => this.addItem("Item 2")}>Add Item 2</button>

<button onClick={() => this.removeItem("Item 1")}>Remove Item 1</button>

<h2>Items in Cart:</h2>

<ul>

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

<li key={index}>{item}</li>

))}

</ul>

</div>

);

}

}

export default CartTracker;

**Exercise 27: Real-Time Stock Tracker**

Create a stock tracker application that fetches stock prices from an API every 10 seconds. Use componentDidMount to start fetching data, componentDidUpdate to compare changes, and componentWillUnmount to clean up.

**Solution**

import React, { Component } from "react";

class StockTracker extends Component {

state = {

stocks: [],

lastUpdated: null,

};

componentDidMount() {

this.fetchStockPrices();

this.stockInterval = setInterval(this.fetchStockPrices, 10000);

}

componentDidUpdate(prevProps, prevState) {

if (prevState.stocks !== this.state.stocks) {

console.log("Stock prices updated:", this.state.stocks);

}

}

componentWillUnmount() {

clearInterval(this.stockInterval);

}

fetchStockPrices = () => {

console.log("Fetching stock prices...");

// Simulating API call

const newPrices = [

{ symbol: "AAPL", price: Math.random() \* 200 },

{ symbol: "GOOGL", price: Math.random() \* 3000 },

{ symbol: "MSFT", price: Math.random() \* 350 },

];

this.setState({

stocks: newPrices,

lastUpdated: new Date().toLocaleTimeString(),

});

};

render() {

const { stocks, lastUpdated } = this.state;

return (

<div>

<h1>Real-Time Stock Tracker</h1>

<p>Last Updated: {lastUpdated || "Fetching..."}</p>

<ul>

{stocks.map((stock, index) => (

<li key={index}>

{stock.symbol}: ${stock.price.toFixed(2)}

</li>

))}

</ul>

</div>

);

}

}

export default StockTracker;

**Exercise 28: Multi-Page Form with Step Tracking**

Build a multi-page form with three steps. Track the current step in the component's state and use componentDidUpdate to log each step transition.

**Solution**

import React, { Component } from "react";

class MultiStepForm extends Component {

state = {

currentStep: 1,

formData: {

name: "",

email: "",

address: "",

},

};

componentDidUpdate(prevProps, prevState) {

if (prevState.currentStep !== this.state.currentStep) {

console.log(`Transitioned to Step ${this.state.currentStep}`);

}

}

nextStep = () => {

if (this.state.currentStep < 3) {

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

}

};

prevStep = () => {

if (this.state.currentStep > 1) {

this.setState((prevState) => ({ currentStep: prevState.currentStep - 1 }));

}

};

handleChange = (field, value) => {

this.setState((prevState) => ({

formData: { ...prevState.formData, [field]: value },

}));

};

render() {

const { currentStep, formData } = this.state;

return (

<div>

<h1>Multi-Step Form</h1>

{currentStep === 1 && (

<div>

<h2>Step 1: Personal Information</h2>

<input

type="text"

placeholder="Name"

value={formData.name}

onChange={(e) => this.handleChange("name", e.target.value)}

/>

</div>

)}

{currentStep === 2 && (

<div>

<h2>Step 2: Contact Information</h2>

<input

type="email"

placeholder="Email"

value={formData.email}

onChange={(e) => this.handleChange("email", e.target.value)}

/>

</div>

)}

{currentStep === 3 && (

<div>

<h2>Step 3: Address</h2>

<input

type="text"

placeholder="Address"

value={formData.address}

onChange={(e) => this.handleChange("address", e.target.value)}

/>

</div>

)}

<div>

<button onClick={this.prevStep} disabled={currentStep === 1}>

Previous

</button>

<button onClick={this.nextStep} disabled={currentStep === 3}>

Next

</button>

</div>

</div>

);

}

}

export default MultiStepForm;

**Exercise 29: Notification Manager**

Build a notification manager component that displays notifications fetched from an API and automatically dismisses them after 5 seconds.

**Solution**

import React, { Component } from "react";

class NotificationManager extends Component {

state = {

notifications: [],

};

componentDidMount() {

this.fetchNotifications();

}

fetchNotifications = () => {

console.log("Fetching notifications...");

const newNotifications = [

{ id: 1, message: "You have a new message!" },

{ id: 2, message: "Your order is ready to be shipped." },

];

this.setState({ notifications: newNotifications }, this.startAutoDismiss);

};

startAutoDismiss = () => {

this.state.notifications.forEach((notification) => {

setTimeout(() => {

this.dismissNotification(notification.id);

}, 5000);

});

};

dismissNotification = (id) => {

this.setState((prevState) => ({

notifications: prevState.notifications.filter((n) => n.id !== id),

}));

};

render() {

const { notifications } = this.state;

return (

<div>

<h1>Notification Manager</h1>

<ul>

{notifications.map((notification) => (

<li key={notification.id}>

{notification.message}

<button onClick={() => this.dismissNotification(notification.id)}>

Dismiss

</button>

</li>

))}

</ul>

</div>

);

}

}

export default NotificationManager;

**Exercise 30: Timer-Based Quiz Application**

Create a quiz component that presents one question at a time, moving to the next question every 15 seconds. If the user answers, it immediately moves to the next question.

**Solution**

import React, { Component } from "react";

class QuizApp extends Component {

state = {

questions: [

{ question: "What is 2 + 2?", answer: "4" },

{ question: "What is the capital of France?", answer: "Paris" },

{ question: "What is React?", answer: "A JavaScript library" },

],

currentQuestionIndex: 0,

timeRemaining: 15,

};

componentDidMount() {

this.startTimer();

}

componentWillUnmount() {

clearInterval(this.timerInterval);

}

startTimer = () => {

this.timerInterval = setInterval(() => {

this.setState((prevState) => ({

timeRemaining: prevState.timeRemaining - 1,

}), () => {

if (this.state.timeRemaining === 0) {

this.nextQuestion();

}

});

}, 1000);

};

nextQuestion = () => {

clearInterval(this.timerInterval);

this.setState((prevState) => ({

currentQuestionIndex:

(prevState.currentQuestionIndex + 1) % prevState.questions.length,

timeRemaining: 15,

}), this.startTimer);

};

render() {

const { questions, currentQuestionIndex, timeRemaining } = this.state;

const currentQuestion = questions[currentQuestionIndex];

return (

<div>

<h1>Quiz App</h1>

<h2>{currentQuestion.question}</h2>

<p>Time Remaining: {timeRemaining} seconds</p>

<button onClick={this.nextQuestion}>Next Question</button>

</div>

);

}

}

export default QuizApp;