REACT NOTE

INTRODUCTION TO REACT

Overview: React is a JavaScript library for building user interfaces, particularly single-page applications (SPAs), by creating reusable UI components. Developed by Facebook (now Meta), it's declarative, component-based, and leverages a virtual DOM for efficient updates. React's core philosophy is to break down UIs into small, self-contained components that manage their own state and can be composed to build complex interfaces.

Key Features:

- Component-Based: Build encapsulated components that manage their own state.

- Virtual DOM: Optimizes performance by minimizing direct DOM manipulations.

- JSX: A syntax extension that blends HTML-like code with JavaScript for intuitive UI design.

- Unidirectional Data Flow: Data flows from parent to child components, making state predictable.

Why It Matters: React simplifies building dynamic, interactive UIs compared to vanilla JavaScript or older frameworks like jQuery. Its ecosystem (e.g., React Router, Redux) and tools like Create React App (CRA) or Vite enable rapid development with a focus on developer experience and performance. For companies, React's reusability and community support reduce development time and maintenance costs.

Setting Up a React Project

Overview: Modern React projects are typically bootstrapped with tools like Create React App (CRA) or Vite, which set up a development environment with Webpack, Babel, and hot-reloading out of the box. I'll focus on Vite for its speed and modern ES modules approach, but I'll mention CRA as an alternative for legacy setups.

Steps to Set Up:

1.Install Node.js: Ensure Node.js (v16+) and npm are installed. Download from [nodejs.org] (https://nodejs.org).

2. Create a Project: Use Vite to scaffold a new React project.

3. Run the App: Start the development server and verify the setup.

4. Add a Simple Component: Create a basic React component to demonstrate JSX and rendering.

Why Vite?: Vite is faster than CRA due to its native ES modules and on-demand compilation. It's the go-to for modern React projects in 2025, especially for teams prioritizing developer experience and build performance.

Practical Example: Setting Up a React App with Vite

Let's create a simple React app with a custom `Welcome` component to display a greeting. This example includes setup commands and a minimal component to confirm everything works.

Step 1: Initialize the Project

Run the following commands in your terminal (ensure Node.js is installed):

Create a new React project with Vite

npm create vite@latest my-react-app -- --template react

Navigate to the project directory

cd my-react-app

Install dependencies

npm install

Start the development server

npm run dev

Vite will create a project structure like this:

my-react-app/

├── node_modules/

├── public/

│ └── vite.svg

├── src/

│ ├── assets/

│ ├── App.jsx

│ ├── main.jsx

│ └── index.css

├── .gitignore

├── index.html

├── package.json

└── vite.config.js

The server will run at `http://localhost:5173` (or similar). Open it in your browser to see the default Vite + React template.

Step 2: Create a Simple Component

Replace the contents of `src/App.jsx` with a custom `Welcome` component to demonstrate React's basics.

```
// src/App.jsx

import React from 'react';

import './App.css'; // Optional: For styling

function Welcome ({name}) {

  return (

    <div>

      <h1>Welcome to React, {name}! < /h1>

      <p>This is a simple React app built with Vite. < /p>

    </div>

  );

}

function App () {

  return (

    <div className="App">

      <Welcome name="Developer" />

    </div>

  );

}

export default App;
```

Update `src/App.css` for basic styling (optional):

```
/* src/App.css */

.App {

  text-align: center;

  padding: 2rem;

  font-family: Arial, sans-serif;

}
```

```
h1 {
  color: #1a73e8;
}
```

Step 3: Verify the Setup

- Save the files, and Vite's hot-reloading will update the browser instantly.

- You should see a page with "Welcome to React, Developer!" and the paragraph below it.

Step 4: Build for Production (Optional)

To test a production build:

```
# Build the app
npm run build
# Preview the production build
npm run preview
```

This creates an optimized `dist/` folder for deployment.

Key Tips for Setup

- *Vite vs. CRA*: Use Vite for faster builds and modern features. CRA is still viable for legacy projects but is slower (`npx create-react-app my-app`).

- *ESLint & Prettier*: Add linting and formatting for code quality:

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

  Initialize ESLint: `npx eslint --init`.

- *TypeScript (Optional)*: For type safety, use Vite's TypeScript template: `npm create vite@latest my-react-app -- --template react-ts`.

- *Folder Structure*: Organize `src/` into `components/`, `pages/`, `hooks/`, and `utils/` for scalability.

- *Environment*: Use `.env` files for API keys (e.g., `VITE_API_KEY=your-key` in Vite).

Why This Matters in a Professional Context

- *Speed*: Vite's fast dev server and hot module replacement (HMR) boost productivity.

- *Scalability*: Starting with a clean setup ensures you can add tools like React Router, Zustand, or Tailwind CSS without friction.

- *Maintainability*: Proper setup with linting and TypeScript prevents tech debt in large teams.

- *Production-Ready*: Vite's optimized builds ensure fast load times, critical for user retention (e.g., Google's Core Web Vitals).

JSX FUNDAMENTALS

Overview: JSX (JavaScript XML) is a syntax extension for JavaScript that allows you to write HTML-like code within JavaScript. It's used in React to describe UI components declaratively, making it easier to visualize the structure of your UI. JSX is transpiled by tools like Babel into plain JavaScript (React.createElement calls) that the browser can understand.

Key Concepts:

- HTML-Like Syntax: Write tags like `<div>` or `<h1>` directly in JavaScript.

- JavaScript Integration: Embed JavaScript expressions inside JSX using curly braces `{}`.

- Component Rendering: JSX allows you to render React components as if they were HTML elements.

- Not HTML: JSX enforces rules like self-closing tags (`<img />`) and camelCase attributes (e.g., `className` instead of `class`).

- Single Root Element: Every JSX block must return a single top-level element (or a Fragment).

Why It Matters: JSX makes React code intuitive and maintainable by combining UI structure with logic in one place. It reduces the mental overhead of switching between HTML and JavaScript, which is common in older frameworks like jQuery or AngularJS. For teams, JSX's declarative nature improves readability and collaboration, especially when paired with TypeScript or linting.

Practical Example: Building a User Card Component with JSX

Let's create a reusable `UserCard` component that displays a user's profile with dynamic data, conditional rendering, and event handling. This example showcases JSX's core features in a production-ready context.

Project Setup

Assuming you have a React project set up with Vite (as covered in the previous topic), create a new file `src/components/UserCard.jsx`. If you're starting fresh, run:

npm create vite@latest my-react-app -- --template react

cd my-react-app

npm install

npm run dev

UserCard Component

Here's a `UserCard` component that demonstrates JSX fundamentals:

- Embedding JavaScript expressions

- Rendering lists

- Conditional rendering
- Handling events
- Using Fragments
- Applying styles with camelCase attributes

// src/components/UserCard.jsx

```
import React from 'react';
import './UserCard.css'; // Optional: For styling
function UserCard({ user, onToggleStatus }) {
  // Sample user data (could come from props or API)
  const { name, email, skills = [], isActive = false } = user;
  return (
    <>
      <div className="user-card">
        <h2>{name.toUpperCase()}</h2> {/* JavaScript expression */}
        <p>Email: {email}</p>
        {/* Conditional rendering */}
        {isActive ? (
          <span style={{ color: 'green' }}>Active</span>
        ) : (
          <span style={{ color: 'red' }}>Inactive</span>
        )}
        {/* List rendering with map */}
        <ul>
          {skills.length > 0 ? (
            skills.map((skill, index) => (
              <li key={index}>{skill}</li> // Unique key for lists
            ))
          ) : (
            <li>No skills listed</li>
          )}
```

```
        </ul>
        {/* Event handling */}
        <button
          onClick={() => onToggleStatus(name)}
          style={{ padding: '8px 16px', cursor: 'pointer' }}
        >
          Toggle Status
        </button>
      </div>
    </>
  );
}
export default UserCard;
```

Styling (Optional)

Add some basic CSS in `src/components/UserCard.css`:

```
.user-card {
  border: 1px solid #ccc;
  border-radius: 8px;
  padding: 16px;
  max-width: 300px;
  margin: 16px auto;
  text-align: center;
  font-family: Arial, sans-serif;
}
h2 {
  color: #1a73e8;
}
ul { list-style: none;
  padding: 0;
}
```

```
li {
  margin: 4px 0;
}
```

Using the Component in App

Update `src/App.jsx` to use the `UserCard` component with sample data and a simple event handler:

```
// src/App.jsx
import React, { useState } from 'react';
import UserCard from './components/UserCard';
import './App.css';
function App() {
  const [users, setUsers] = useState([
    {
      name: 'Alice Smith',
      email: 'alice@example.com',
      skills: ['React', 'JavaScript', 'CSS'],
      isActive: true,
    },
    {
      name: 'Bob Johnson',
      email: 'bob@example.com',
      skills: [],
      isActive: false,
    },
  ]);



  const handleToggleStatus = (name) => {
    setUsers(
      users.map((user) =>
        user.name === name ? { ...user, isActive: !user.isActive } : user
```

```
      )
    );
  };

  return (
    <div className="App">
      <h1>User Profiles</h1>
      {users.map((user) => (
        <UserCard
          key={user.email} // Unique key for list rendering
          user={user}
          onToggleStatus={handleToggleStatus}
        />
      ))}
    </div>
  );
}
export default App;
```

Update `src/App.css` for basic layout:

```
.App {
  text-align: center;
  padding: 2rem;
  font-family: Arial, sans-serif;
}

h1 {
  color: #333;
}
```

Running the App

Run `npm run dev` and visit `http://localhost:5173`. You'll see two user cards with names, emails, skills, status indicators, and a toggle button. Clicking the button toggles the user's status between "Active" and "Inactive."

Breaking Down JSX Features in the Example

1. JSX Tags and Attributes:

- Used HTML-like tags (`<div>`, `<h2>`, `<button>`).

- Attributes like `className` and `style` use camelCase, not HTML's `class` or `style`.

- Inline styles are objects (e.g., `style={{ color: 'green' }}`).

2. JavaScript Expressions:

- `{name.toUpperCase()}` embeds a JavaScript expression to transform the name.

- `{skills.length > 0 ? ... : ...}` uses a ternary operator for conditional rendering.

3. List Rendering:

- `skills.map()` renders a dynamic list of `<li>` elements.

- The `key` prop (`key={index}`) ensures React efficiently updates the DOM.

4. Fragments:

- `<></>` wraps the JSX without adding extra DOM nodes, unlike a `<div>`.

5. Event Handling:

- `onClick={() => onToggleStatus(name)}` binds a click event to a function, passing the user's name.

- Arrow functions prevent immediate invocation during render.

6. Props:

- The `user` and `onToggleStatus` props pass data and behavior to the component.

Best Practices for JSX

- Always Use Keys in Lists: Prevent rendering bugs and improve performance (e.g., `key={user.email}`).

- Self-Closing Tags: Tags like `<img />` or `<br />` must be self-closed.

- Avoid Inline Logic: Move complex logic outside JSX for readability (e.g., compute values before the return).

- **Sanitize Inputs:** If rendering user input, use libraries like `DOMPurify` to prevent XSS attacks.

- **Use Fragments:** Avoid unnecessary `<div>` wrappers with `<React.Fragment>` or `<></>`.

- **CamelCase for Attributes:** Use `className`, `htmlFor`, etc., instead of HTML attributes.

- **Type Safety (TypeScript):** Define prop types for better maintainability:

```
import PropTypes from 'prop-types';
UserCard.propTypes = {
  user: PropTypes.shape({
    name: PropTypes.string.isRequired,
    email: PropTypes.string.isRequired,
    skills: PropTypes.arrayOf(PropTypes.string),
    isActive: PropTypes.bool,
  }).isRequired,
  onToggleStatus: PropTypes.func.isRequired,
};
```

Or use TypeScript interfaces for stricter typing.

Why JSX Matters in a Professional Context

- **Readability:** JSX's HTML-like syntax makes UI code intuitive, reducing onboarding time for new developers.

- **Reusability:** Components like `UserCard` can be reused across the app, saving development time.

- **Maintainability:** JSX's declarative nature makes debugging easier (e.g., React DevTools shows component hierarchy).

- **Performance:** JSX compiles to optimized `React.createElement` calls, leveraging the virtual DOM for fast updates.

- **Team Collaboration:** Standardized JSX patterns (e.g., consistent prop naming) improve code reviews and scalability.

Common Pitfalls and How to Avoid Them

- **Forgetting Keys in Lists:** Always provide a unique `key` prop to avoid React warnings and rendering issues.

- **Overusing Inline Styles:** Use CSS modules or styled-components for maintainable styling.

- Complex JSX: Keep JSX lean by extracting logic to functions or hooks:

```
const getStatusColor = (isActive) => (isActive ? 'green' : 'red');
<span style={{ color: getStatusColor(isActive) }}>{isActive ? 'Active' : 'Inactive'}</span>
```

- **Not Closing Tags:** Ensure all tags are closed (e.g., `<input />`).

Awesome, let's dive into React Components: Functional vs. Class Components as the next topic. Since you've already covered Introduction to React & Setup and JSX Fundamentals, and we just discussed React Components in general, I'll focus specifically on comparing Functional Components and Class Components. I'll provide a clear definition, a detailed comparison, practical examples, and insights from a senior React developer's perspective to help you understand their differences, use cases, and best practices.

FUNCTIONAL VS CLASS COMPONENTS

Definition

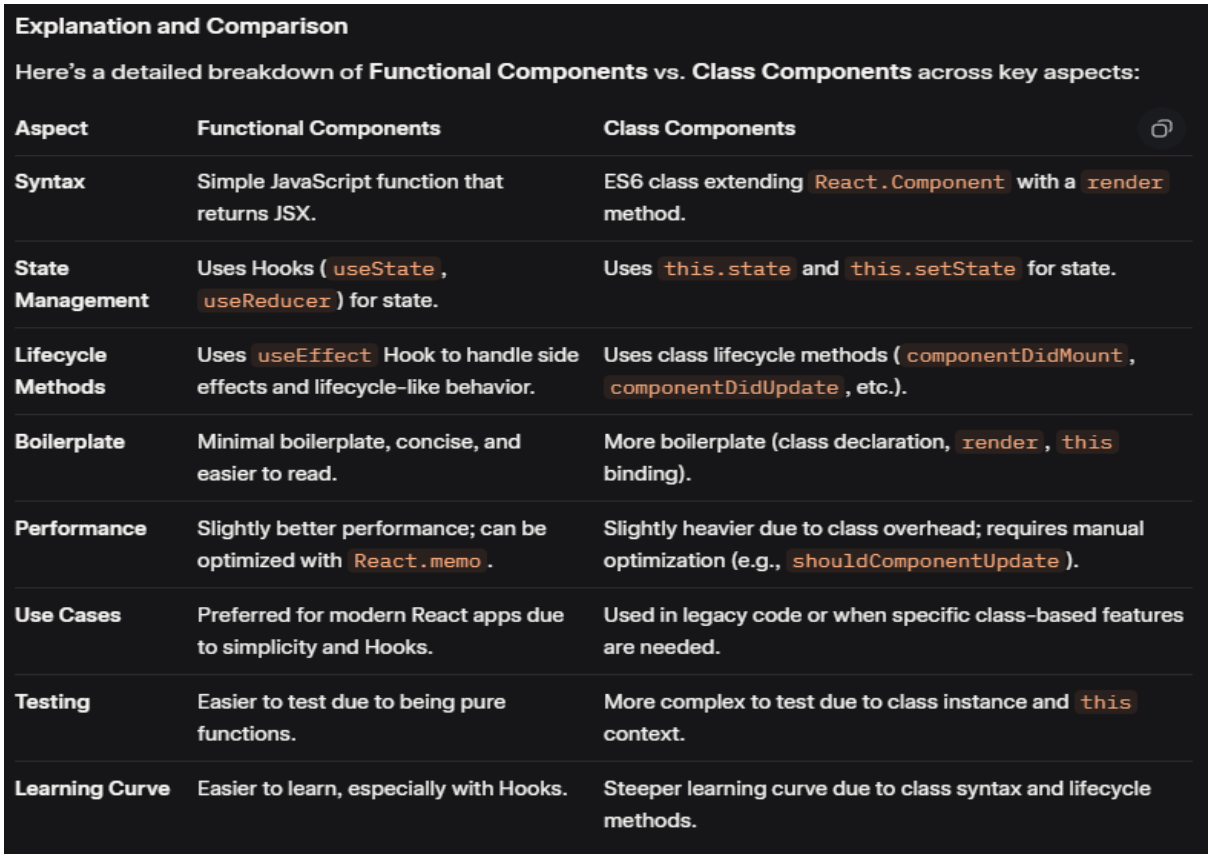
- ***Functional Components*:** These are JavaScript functions that return JSX to describe a piece of the UI. They are lightweight, concise, and have become the standard in modern React development, especially since the introduction of Hooks in React 16.8, which allow functional components to manage state and side effects.

- ***Class Components*:** These are ES6 classes that extend `React.Component` and include a `render` method to return JSX. They were the primary way to build stateful components before Hooks but are now less common due to their verbosity and complexity.

Both types of components can receive props, render UI, and be composed to build complex interfaces. The key differences lie in their syntax, features, and how they handle state and lifecycle methods.

Explanation and Comparison

Here's a detailed breakdown of Functional Components vs. Class Components across key aspects:



Explanation and Comparison

Here's a detailed breakdown of **Functional Components** vs. **Class Components** across key aspects:

Aspect	Functional Components	Class Components
Syntax	Simple JavaScript function that returns JSX.	ES6 class extending `React.Component` with a `render` method.
State Management	Uses Hooks (`useState` , `useReducer`) for state.	Uses `this.state` and `this.setState` for state.
Lifecycle Methods	Uses `useEffect` Hook to handle side effects and lifecycle-like behavior.	Uses class lifecycle methods (`componentDidMount` , `componentDidUpdate` , etc.).
Boilerplate	Minimal boilerplate, concise, and easier to read.	More boilerplate (class declaration, `render` , `this` binding).
Performance	Slightly better performance; can be optimized with `React.memo` .	Slightly heavier due to class overhead; requires manual optimization (e.g., `shouldComponentUpdate`).
Use Cases	Preferred for modern React apps due to simplicity and Hooks.	Used in legacy code or when specific class-based features are needed.
Testing	Easier to test due to being pure functions.	More complex to test due to class instance and `this` context.
Learning Curve	Easier to learn, especially with Hooks.	Steeper learning curve due to class syntax and lifecycle methods.

Key Points

- *Functional Components*:

- Introduced as "stateless" components but became fully featured with Hooks.
- Use `useState` for state and `useEffect` for side effects (e.g., fetching data, subscriptions).
- More concise and align better with JavaScript's functional programming paradigm.
- Can be memoized with `React.memo` to prevent unnecessary re-renders.

- Example Hook usage:

```
import React, { useState, useEffect } from 'react';
function Counter() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `Count: ${count}`;
```

```
  }, [count]);
  return (
    <button onClick={() => setCount(count + 1)}>
      Clicked {count} times
    </button>
  );
}
```

- *Class Components*:

- Rely on `this` for state, props, and methods, which can lead to binding issues (e.g., `this.handleClick = this.handleClick.bind(this)`).

- Have explicit lifecycle methods like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.

- More verbose and prone to errors in complex components.

- Example lifecycle usage:

```
import React, { Component } from 'react';
class Counter extends Component {
  state = { count: 0 };
  componentDidUpdate() {
    document.title = `Count: ${this.state.count}`;
  }
  handleClick = () => {
    this.setState({ count: this.state.count + 1 });
  };
  render() {
    return (
      <button onClick={this.handleClick}>
        Clicked {this.state.count} times
      </button>
    );
  }
}
```

- Why Prefer Functional Components?

- ***Simplicity*:** Less code, no `this` binding issues, and easier to understand.

- ***Hooks*:** Provide a more flexible and reusable way to handle state and side effects compared to lifecycle methods.

- ***Community Shift*:** Modern React libraries, tutorials, and frameworks (e.g., Next.js) focus on functional components.

- ***Future-Proof*:** React's roadmap emphasizes functional components and Hooks.

- When to Use Class Components?

- Legacy codebases that haven't been refactored.

- Rare cases requiring specific class-based features, like `ErrorBoundary` (though Hooks like `useErrorBoundary` are emerging).

- **Example of an `ErrorBoundary` (only possible with class components):**

```
class ErrorBoundary extends React.Component {
  state = { hasError: false };
  static getDerivedStateFromError(error) {
    return { hasError: true };
  }
  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}
```

Example 1: Functional Component with State and Effect

This example shows a functional component using `useState` and `useEffect` to create a counter that updates the document title.

// Counter.js

```
import React, { useState, useEffect } from 'react';
function Counter({ initialCount = 0 }) {
  const [count, setCount] = useState(initialCount);
  // Equivalent to componentDidMount + componentDidUpdate
  useEffect(() => {
    document.title = `Count: ${count}`;
    console.log('Effect ran');
    // Cleanup (equivalent to componentWillUnmount)
    return () => {
      console.log('Cleanup ran');
    };
  }, [count]); // Runs when `count` changes
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
export default Counter;
```

Usage in App:

```
// App.js
import React from 'react';
import Counter from './Counter';
function App() {
  return (
    <div>
      <Counter initialCount={5} />
    </div>
```

```
  );
}
export default App;
```

Explanation:

- `useState` manages the `count` state, initialized with `initialCount` prop.

- `useEffect` updates the document title whenever `count` changes and logs a message. The cleanup function runs before the next effect or when the component unmounts.

- The component is concise, readable, and handles state and side effects without class boilerplate.

Example 2: Class Component with State and Lifecycle

Here's the same counter implemented as a class component.

```
// Counter.js
import React, { Component } from 'react';
class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: props.initialCount || 0,
    };
  }
  componentDidMount() {
    document.title = `Count: ${this.state.count}`;
    console.log('Component mounted');
  }
  componentDidUpdate() {
    document.title = `Count: ${this.state.count}`;
    console.log('Component updated');
  }
  componentWillUnmount() {
    console.log('Component will unmount');}
```

```
  handleIncrement = () => {
    this.setState({ count: this.state.count + 1 });
  };
  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.handleIncrement}>Increment</button>
      </div>
    );
  }
}
export default Counter;
```

Usage: Same as above in `App.js`.

Explanation:

- The class uses `this.state` and `this.setState` for state management.

-Lifecycle methods (`componentDidMount`, `componentDidUpdate`, `componentWillUnmount`) handle side effects, equivalent to `useEffect`.

- The code is more verbose, requires `this` binding for methods (e.g., `handleIncrement`), and uses a `constructor` for initialization.

Example 3: Converting a Class Component to Functional

Let's take a simple class component and convert it to a functional component to highlight the differences.

Class Component:

```
// Timer.js
import React, { Component } from 'react';
class Timer extends Component {
  state = { seconds: 0 };
```

```
  componentDidMount() {
    this.timerID = setInterval(() => {
      this.setState({ seconds: this.state.seconds + 1 });
    }, 1000);
  }
  componentWillUnmount() {
    clearInterval(this.timerID);
  }
  render() {
    return <h2>Seconds Elapsed: {this.state.seconds}</h2>;
  }
}
export default Timer;
```

Converted to Functional Component:

```
// Timer.js
import React, { useState, useEffect } from 'react';
function Timer() {
  const [seconds, setSeconds] = useState(0);
  useEffect(() => {
    const timerID = setInterval(() => {
      setSeconds((prev) => prev + 1);
    }, 1000);
    // Cleanup
    return () => clearInterval(timerID);
  }, []); // Empty dependency array: runs once on mount
  return <h2>Seconds Elapsed: {seconds}</h2>;
}
export default Timer;
```

Key Differences:

- The functional component uses `useState` instead of `this.state`.

- `useEffect` replaces `componentDidMount` and `componentWillUnmount` with a single Hook.

- No `this` keyword, making the code cleaner and less error-prone.

- The functional version uses a functional update (`prev`) in `setSeconds` for safer state updates.

Senior-Level Insights

- When to Choose Functional Components:

- Use functional components for all new React code unless you're maintaining legacy code.

- Hooks like `useState`, `useEffect`, `useContext`, and custom Hooks make functional components more powerful and reusable than class components.

- Functional components align with React's future direction and are easier to optimize with tools like `React.memo`.

- When to Use Class Components:

- Legacy codebases that rely heavily on class components.

- Specific cases like `ErrorBoundary`, which currently requires a class component (though Hooks-based alternatives are emerging).

- If your team or project has strict requirements for class-based components (rare).

- Performance Considerations:

- Functional components can be wrapped with `React.memo` to prevent unnecessary re-renders:

```
export default React.memo(MyComponent);
```

- Class components require implementing `shouldComponentUpdate` or extending `React.PureComponent` for similar optimization, which is more manual and error-prone.

- Best Practices:

- Avoid mixing functional and class components in the same codebase for consistency.

- Use TypeScript or PropTypes to enforce prop types in both component types.

- For functional components, leverage custom Hooks to extract reusable logic (e.g., `useTimer` for the timer logic above).

- Keep components small and focused, regardless of type, to improve maintainability.

- Migration Tip:

- When converting class components to functional, map lifecycle methods to Hooks:

  - `componentDidMount` → `useEffect(() => {}, [])`

  - `componentDidUpdate` → `useEffect(() => {}, [dependencies])`

  - `componentWillUnmount` → `useEffect(() => { return () => cleanup; }, [dependencies])`

- Use tools like React's official migration guides or automated refactoring tools (e.g., `jscodeshift`) for large codebases.

VIRTUAL DOM, REACT FIBER, AND RECONCILIATION

1. Virtual DOM

Definition

The Virtual DOM is an in-memory tree of JavaScript objects mirroring the real DOM, used by React to optimize UI updates by minimizing direct DOM manipulations.

Explanation

- Purpose: The real DOM is slow to update due to reflows/repaints. The Virtual DOM enables React to compute changes in memory and apply only necessary updates.

- Mechanism: React creates a Virtual DOM tree on each render, compares it with the previous tree (via reconciliation), and updates the real DOM efficiently.

- Benefits: Faster updates, declarative UI, and simplified development by abstracting DOM operations.

- Senior Insights:

- Not a performance silver bullet; optimize with `React.memo`, `useMemo`, or `useCallback` to reduce re-renders.

- Avoid manual DOM manipulation (e.g., `document.querySelector`) to maintain Virtual DOM consistency.

2. Reconciliation

Definition

Reconciliation is React's algorithm for comparing the new and previous Virtual DOM trees to identify minimal changes needed for real DOM updates.

Explanation

- Process: On state/prop changes, React generates a new Virtual DOM tree, diffs it with the old tree, and applies batched DOM updates.

- Heuristics:

- Same element type: Updates props/state, reuses DOM node.

- Different type: Replaces entire subtree.

- `key` prop: Optimizes list updates by matching elements.

- Performance: Fast due to in-memory diffing and heuristics. Use unique `key` props for lists to avoid inefficiencies.

- Senior Insights:

- Stable `key` props are critical for list performance.

- Optimize with `React.memo` to skip unnecessary reconciliations.

- Deep component trees can slow reconciliation; flatten or use virtualized rendering (e.g., `react-window`).

3. React Fiber

Definition

React Fiber is React's modern reconciliation engine (since React 16), enabling incremental rendering and priority scheduling for better performance.

Explanation

- Purpose: Replaces the synchronous stack reconciler, which could freeze UIs in complex apps. Fiber processes updates in small, interruptible chunks.

- Features:

- Incremental Rendering: Breaks reconciliation into chunks, pausing for urgent tasks (e.g., user input).

- Priority Scheduling: Prioritizes high-priority updates (e.g., animations) over low-priority ones (e.g., data fetching).

- Concurrent Features: Supports Concurrent Mode, Suspense, and Time Slicing.

- Mechanism: Uses a fiber tree (objects representing components/elements) to track and schedule work, committing changes to the DOM in batches.

- Senior Insights:

- Fiber is internal; developers benefit indirectly via responsiveness.

- Leverage Concurrent Mode/Suspense for advanced use cases.

- Optimize with memoization to reduce Fiber's workload.

- Use React Profiler to analyze rendering and ensure Fiber's efficiency.

PROPS & PROP TYPES

Props (short for properties) are a mechanism in React for passing data from a parent component to a child component. They are immutable, meaning the child component cannot modify the props it receives; they are read-only. Props allow components to be reusable and dynamic by enabling customization based on the data passed to them.

Prop Types is a type-checking mechanism provided by the `prop-types` library to validate the type and structure of props a component expects. This helps catch bugs early during development by ensuring props conform to expected types (e.g., string, number, array) and can enforce required props.

Key Points:

- Props are passed as attributes to a component, similar to HTML attributes.
- Props can be any JavaScript value: strings, numbers, objects, arrays, functions, etc.
- Use default props to provide fallback values when props are not passed.
- Prop Types is optional but highly recommended for catching errors in development.
- In modern React (since React 15.5), Prop Types is a separate package (`prop-types`).
- For production apps, consider using TypeScript for static type checking instead of Prop Types.

#Installation

To use Prop Types, install the `prop-types` package:

npm install prop-types

Example

Let's create a reusable `UserCard` component that displays user information passed via props, with Prop Types to enforce data types and required props.

1. UserCard Component (`UserCard.jsx`):

```
import PropTypes from 'prop-types';

function UserCard({ name, age, isActive, hobbies, onGreet }) {
  return (
    <div style={{ border: '1px solid #ccc', padding: '16px', margin: '8px' }}>
      <h2>{name}</h2>
      <p>Age: {age}</p>
```

```
      <p>Status: {isActive ? 'Active' : 'Inactive'}</p>
      <p>Hobbies: {hobbies.join(', ')}</p>
      <button onClick={() => onGreet(name)}>Greet User</button>
    </div>
  );
}

// Define Prop Types
UserCard.propTypes = {
  name: PropTypes.string.isRequired, // Required string
  age: PropTypes.number.isRequired, // Required number
  isActive: PropTypes.bool, // Optional boolean
  hobbies: PropTypes.arrayOf(PropTypes.string).isRequired, // Required array of strings
  onGreet: PropTypes.func.isRequired, // Required function
};

// Define default props
UserCard.defaultProps = {
  isActive: false, // Default value if not provided
};
export default UserCard;
```

2. Parent Component (`App.jsx`):

```
import UserCard from './UserCard';
function App() {
  const handleGreet = (name) => {
    alert(`Hello, ${name}!`);
  };
  return (
    <div>
```

```
      <h1>User Profiles</h1>
      <UserCard
        name="Alice"
        age={25}
        isActive={true}
        hobbies={['Reading', 'Hiking']}
        onGreet={handleGreet}
      />
      <UserCard
        name="Bob"
        age={30}
        hobbies={['Gaming', 'Cooking']}
        onGreet={handleGreet}
      />
    </div>
  );
}
export default App;
```

3. Output in Browser:

- Two cards will display:

  - Alice: Age 25, Active, Hobbies: Reading, Hiking, with a "Greet User" button.

  - Bob: Age 30, Inactive (default `isActive`), Hobbies: Gaming, Cooking, with a "Greet User" button.

- Clicking the "Greet User" button triggers an alert (e.g., "Hello, Alice!").

4. Prop Types Validation:

- If you pass incorrect props (e.g., `age="30"` as a string instead of a number), you'll see a warning in the console during development:

  Warning: Failed prop type: Invalid prop `age` of type `string` supplied to `UserCard`, expected `number`.

- If a required prop like `name` is missing, you'll get:

Warning: Failed prop type: The prop `name` is marked as required in `UserCard`, but its value is `undefined`.

Best Practices

1. Always define Prop Types in development to catch errors early, especially in team projects.

2. Use `isRequired` for props that are essential for the component to function correctly.

3. Provide `defaultProps` for optional props to avoid runtime errors (e.g., `undefined` values).

4. Destructure props in the function signature for cleaner code (e.g., `({ name, age })`).

5. For complex projects, consider TypeScript for stronger type safety over Prop Types.

6. Keep props minimal and relevant to maintain component reusability.

7. Use descriptive prop names to improve code readability.

Insights

- Props vs State: Props are for parent-to-child data flow, while state is for internal component data that changes over time. Avoid duplicating prop data in state unless necessary.

- Performance: Passing large objects or functions as props can trigger unnecessary re-renders. Use `React.memo` or `useCallback` to optimize if needed.

- Prop Drilling: If props need to pass through multiple component layers, consider using Context API or state management libraries like Redux.

- Type Safety: Prop Types is great for small projects, but TypeScript is preferred for large-scale apps due to its compile-time type checking and IDE support.

Common Prop Types

Here are some commonly used Prop Types:

```
import PropTypes from 'prop-types';

Component.propTypes = {
  stringProp: PropTypes.string,
  numberProp: PropTypes.number,
  boolProp: PropTypes.bool,
  arrayProp: PropTypes.array,
  objectProp: PropTypes.object,
  funcProp: PropTypes.func,
  nodeProp: PropTypes.node, // Anything that can be rendered: string, number, JSX
```

```
  elementProp: PropTypes.element, // A React element
  specificType: PropTypes.oneOfType([PropTypes.string, PropTypes.number]), // Union type
  arrayOf: PropTypes.arrayOf(PropTypes.string), // Array of specific type
  objectOf: PropTypes.objectOf(PropTypes.number), // Object with values of specific type
  shapeProp: PropTypes.shape({
    id: PropTypes.number.isRequired,
    name: PropTypes.string,
  }), // Object with specific shape
};
```

Why Use Prop Types?

- Debugging: Catches type mismatches during development.

- Documentation: Serves as self-documenting code, making it clear what props a component expects.

- Team Collaboration: Ensures team members pass correct data to components.

When to Skip Prop Types?

- In small, solo projects where type errors are less likely.

- When using TypeScript, as it provides stronger type checking.

- In performance-critical apps, as Prop Types adds a slight overhead (though it's stripped in production builds).

STATE MANAGEMENT BASICS

State in React is a built-in object that holds data or information about a component's current situation, which can change over time due to user interactions, API calls, or other events. Unlike props, which are passed from parent to child and are immutable, state is managed within the component and can be updated using specific functions provided by React. State changes trigger re-renders to reflect the updated UI.

In React, state management differs between functional and class components:

- Functional Components: Use the `useState` hook (introduced in React 16.8) to manage state.

- Class Components: Use `this.state` and `this.setState` for state management.

Since functional components with hooks are the modern standard, I'll focus primarily on `useState`, but I'll also include a class-based example for completeness.

Key Points:

- State is local to a component unless shared via props or Context.

- Always update state using the provided setter function (e.g., `setState` or `setCount` from `useState`) to ensure React re-renders the component.

- State updates are asynchronous, so use callback functions when the new state depends on the previous state.

- Avoid storing redundant data in state (e.g., data derivable from props or other state).

- For complex state logic, consider `useReducer` or external state management libraries like Redux or Zustand.

Example: Counter App with State

Let's create a simple counter application to demonstrate state management in both functional and class components.

1. Functional Component with `useState`

Counter.jsx:

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

function Counter() {
  // Declare state variable 'count' with initial value 0
  const [count, setCount] = useState(0);

  // Event handlers
  const increment = () => {
    setCount(prevCount => prevCount + 1); // Use callback for safe updates
  };
  const decrement = () => {
    setCount(prevCount => prevCount - 1);
  };
  const reset = () => {
    setCount(0);
  };
  return (
    <div style={{ padding: '16px', textAlign: 'center' }}>
      <h2>Counter: {count}</h2>
      <button onClick={increment}>Increment</button>
```

```
      <button onClick={decrement}>Decrement</button>
      <button onClick={reset}>Reset</button>
    </div>
  );
}
export default Counter;
```

2. Class Component with `this.state`

CounterClass.jsx:

```
import React, { Component } from 'react';
class CounterClass extends Component {
  // Initialize state
  state = {
    count: 0,
  };
  // Event handlers
  increment = () => {
    this.setState(prevState => ({ count: prevState.count + 1 }));
  };

  decrement = () => {
    this.setState(prevState => ({ count: prevState.count - 1 }));
  };
  reset = () => {
    this.setState({ count: 0 });
  };
  render() {
    return (
      <div style={{ padding: '16px', textAlign: 'center' }}>
        <h2>Counter: {this.state.count}</h2>
```

```
        <button onClick={this.increment}>Increment</button>
        <button onClick={this.decrement}>Decrement</button>
        <button onClick={this.reset}>Reset</button>
      </div>
    );
  }
}
export default CounterClass;
```

3. Parent Component (`App.jsx`)

```
import Counter from './Counter';
import CounterClass from './CounterClass';
function App() {
  return (
    <div>
      <h1>State Management Demo</h1>
      <h3>Functional Component Counter</h3>
      <Counter />
      <h3>Class Component Counter</h3>
      <CounterClass />
    </div>
  );
}
export default App;
```

4. Output in Browser

- Two counters will display: one using `useState` (functional) and one using `this.state` (class).

- Each counter shows the current count and has buttons to increment, decrement, or reset the count.

- Clicking the buttons updates the respective counter’s state, triggering a re-render to display the new count.

Key Concepts in the Example

1. `useState` Hook (Functional):

- `useState(0)` initializes the `count` state with a value of `0`.

- Returns an array with the state variable (`count`) and its setter function (`setCount`).

- Use the callback form (`prevCount => prevCount + 1`) for updates that depend on the previous state to avoid issues with asynchronous updates.

2. `this.state` and `setState` (Class):

- State is initialized in the `state` object.

- `setState` updates state and triggers a re-render.

- Use the callback form (`prevState => ({ count: prevState.count + 1 })`) for safe updates.

3. Why Use Callback for Updates?

- State updates are asynchronous, so direct updates like `setCount(count + 1)` may not reflect the latest state in complex scenarios (e.g., multiple updates in a single event).

- The callback ensures you're working with the latest state.

Best Practices

1. Initialize State Properly: Use meaningful initial values (e.g., `0` for a counter, `""` for a text input).

2. Keep State Minimal: Only store data that directly affects the UI. Derive computed values outside of state.

- Example: If you need `isEven` based on `count`, compute it like `const isEven = count % 2 === 0` instead of storing it in state.

3. Use Functional Updates: Always use the callback form for state updates that depend on the previous state.

4. Avoid Direct State Mutation: Never modify state directly (e.g., `this.state.count++` in class components or `count++` in functional components). Always use the setter function.

5. Group Related State: For related state variables, consider using an object or `useReducer` for better organization.

```
const [form, setForm] = useState({ name: '', email: '' });
setForm(prev => ({ ...prev, name: 'New Name' }));
```

6. Functional Components Preferred: Use `useState` in functional components unless you have a specific reason to use class components (e.g., legacy code).

7. Debugging: Use React Developer Tools to inspect state changes.

Insights

- When to Use State: Use state for data that changes over time and affects the UI, such as form inputs, counters, or toggles. For static or passed data, use props.

- Complex State: For complex state logic (e.g., multiple interdependent values), consider `useReducer` instead of multiple `useState` calls.

```
const [state, dispatch] = useReducer(reducer, { count: 0 });
```

- State vs. Props: State is internal and mutable; props are external and immutable. Pass state down as props to child components when needed.

- Performance: Avoid unnecessary state updates to prevent excessive re-renders. Use `React.memo` or `useMemo` for optimization if state changes cause performance issues.

- Beyond Component State: For global state (shared across components), consider Context API, Redux, or Zustand. For example, if multiple components need access to a user's authentication status, use Context instead of prop drilling.

Common Pitfalls

1. Mutating State Directly:

```
// Wrong
const [items, setItems] = useState([]);
items.push('new item'); // Direct mutation (won't trigger re-render)
// Correct
setItems([...items, 'new item']);
```

2. Overusing State:

- Don't store computed values in state. For example, if you have `firstName` and `lastName` in state, compute `fullName` outside state:

```
const fullName = `${firstName} ${lastName}`;
```

3. Ignoring Asynchronicity:

- State updates may not reflect immediately. Use `useEffect` to react to state changes if needed:

```
useEffect(() => {
  console.log(`Count updated to: ${count}`);
}, [count]);
```

HOOKS: USESTATE & USEEFFECT

Hooks are functions that let you use state and other React features in functional components, introduced in React 16.8 to replace class-based state and lifecycle methods. They make code more concise, reusable, and easier to understand compared to class components.

- useState: A hook for managing state in functional components. It allows you to declare a state variable and a setter function to update it, triggering a re-render when the state changes.

- useEffect: A hook for handling side effects in functional components, such as data fetching, subscriptions, or DOM manipulations. It replaces lifecycle methods like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` in class components.

Key Points:

- `useState` is used for local component state (e.g., form inputs, toggles, counters).

- `useEffect` runs after every render by default but can be controlled with a dependency array.

- Hooks can only be used in functional components or custom hooks (not in class components or regular JavaScript functions).

- Hooks follow the "Rules of Hooks": call them at the top level of your component (not inside loops, conditions, or nested functions) and only in React functional components or custom hooks.

Example: Todo List with Data Fetching

Let's create a simple todo list application that uses `useState` to manage todos and `useEffect` to fetch todos from a mock API (simulated with a Promise). This example demonstrates both hooks working together.

1. Todo Component (`TodoList.jsx`)

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

function TodoList() {
  // State for todos and loading status
  const [todos, setTodos] = useState([]);
```

```
const [newTodo, setNewTodo] = useState('');
const [isLoading, setIsLoading] = useState(true);
// Mock API function to fetch todos
const fetchTodos = async () => {
  try {
    // Simulate API call
    const response = await new Promise(resolve =>
      setTimeout(() => resolve([
        { id: 1, text: 'Learn React Hooks', completed: false },
        { id: 2, text: 'Build a Todo App', completed: false },
      ]), 1000)
    );
    setTodos(response);
    setIsLoading(false);
  } catch (error) {
    console.error('Error fetching todos:', error);
    setIsLoading(false);
  }
};
// useEffect to fetch todos on component mount
useEffect(() => {
  fetchTodos();
  // Cleanup (optional, e.g., for aborting fetch in real APIs)
  return () => {
    console.log('Cleaning up useEffect');
  };
}, []); // Empty dependency array: runs only on mount
// Handle adding a new todo
const addTodo = () => {
  if (newTodo.trim() === '') return;
```

```
    const newTodoItem = {
      id: todos.length + 1,
      text: newTodo,
      completed: false,
    };
    setTodos([...todos, newTodoItem]);
    setNewTodo(''); // Clear input
  };
  // Handle toggling todo completion
  const toggleTodo = (id) => {
    setTodos(
      todos.map(todo =>
        todo.id === id ? { ...todo, completed: !todo.completed } : todo
      )
    );
  };
  // Handle input change
  const handleInputChange = (e) => {
    setNewTodo(e.target.value);
  };
  return (
    <div style={{ padding: '16px', maxWidth: '600px', margin: '0 auto' }}>
      <h1>Todo List</h1>
      <div>
        <input
          type="text"
          value={newTodo}
          onChange={handleInputChange}
          placeholder="Add a new todo"
        />
```

```
        <button onClick={addTodo} style={{ marginLeft: '8px' }}>
          Add Todo
        </button>
      </div>
      {isLoading ? (
        <p>Loading todos...</p>
      ) : (
        <ul style={{ listStyle: 'none', padding: 0 }}>
          {todos.map(todo => (
            <li
              key={todo.id}
              style={{
                textDecoration: todo.completed ? 'line-through' : 'none',
                cursor: 'pointer',
                margin: '8px 0',
              }}
              onClick={() => toggleTodo(todo.id)}
            >
              {todo.text}
            </li>
          ))}
        </ul>
      )}
    </div>
  );
}
export default TodoList;
```

2. Parent Component (`App.jsx`)

```
import TodoList from './TodoList';
function App() {
  return (
    <div>
      <h1>React Hooks Demo</h1>
      <TodoList />
    </div>
  );
}
export default App;
```

3. Output in Browser

- On load, a “Loading todos…” message displays for 1 second (simulated API delay).
- After loading, two todos appear: “Learn React Hooks” and “Build a Todo App.”
- Users can:
  - Add new todos by typing in the input and clicking “Add Todo.”
  - Click a todo to toggle its completion status (strikes through when completed).
- The UI updates reactively as state changes.

Key Concepts in the Example

1. useState:

- Manages three pieces of state:
  - `todos`: Array of todo objects.
  - `newTodo`: String for the input field.
  - `isLoading`: Boolean to show a loading state.
- Updates are triggered by `setTodos`, `setNewTodo`, and `setIsLoading`.
- Example: `setTodos([...todos, newTodoItem])` spreads the existing todos and adds a new one, ensuring immutability.

2. useEffect:

- Runs `fetchTodos` when the component mounts (empty dependency array `[]`).

- Simulates an API call with a 1-second delay, updating `todos` and `isLoading` on success.

- Includes a cleanup function (logged for demonstration), which runs when the component unmounts or before the effect runs again (if dependencies change).

- Dependency array (`[]`) ensures the effect runs only once on mount, mimicking `componentDidMount`.

Best Practices

1. useState:

- Initialize with meaningful values (e.g., `[]` for arrays, `""` for strings).

- Use functional updates for state changes that depend on the previous state:

```
setTodos(prevTodos => [...prevTodos, newTodoItem]);
```

- Keep state minimal and avoid storing derivable data (e.g., don't store `todosCount` when you can use `todos.length`).

2. useEffect:

- Always include a dependency array to control when the effect runs:

  - `[]`: Runs once on mount.

  - `[dep1, dep2]`: Runs on mount and when `dep1` or `dep2` changes.

  - No array: Runs after every render (avoid unless necessary).

- Return a cleanup function for effects that need it (e.g., canceling timers, unsubscribing from WebSockets):

```
useEffect(() => {
  const timer = setInterval(() => console.log('Tick'), 1000);
  return () => clearInterval(timer); // Cleanup
}, []);
```

- Handle errors in async effects using try-catch.

3. General:

- Follow the Rules of Hooks: Call hooks at the top level, not inside loops or conditions.

- Use descriptive state variable names (e.g., `todos` instead of `data`).

- Combine `useState` and `useEffect` for dynamic UIs, like fetching data and updating the UI based on user input.

- For complex state logic, consider `useReducer` instead of multiple `useState` calls.

Insights

- useState vs. this.state: `useState` is simpler and avoids the verbosity of `this.setState` in class components. It also allows multiple independent state variables without merging issues.

- useEffect vs. Lifecycle Methods:

- `useEffect` with `[]` ≈ `componentDidMount`.

- `useEffect` with `[dep]` ≈ `componentDidUpdate` for `dep`.

- Cleanup function ≈ `componentWillUnmount`.

- Unlike class lifecycle methods, `useEffect` can handle multiple side effects in one component.

- Common Use Cases for useEffect:

- Fetching data from APIs.

- Setting up subscriptions (e.g., WebSockets, event listeners).

- Updating the document title or other DOM side effects:

```
useEffect(() => {
  document.title = `Todos: ${todos.length}`;
}, [todos.length]);
```

- Performance: Avoid unnecessary re-renders by memoizing callbacks (`useCallback`) or values (`useMemo`) if passed to `useEffect` dependencies.

- Async in useEffect: Since `useEffect` can't be async directly, wrap async logic in a function:

```
useEffect(() => {
  async function fetchData() {
    const data = await fetch('/api/data');
    setData(data);
  }
  fetchData();
}, []);
```

Common Pitfalls

1. Missing Dependencies in useEffect:

- Omitting dependencies in the dependency array can cause stale data or infinite loops:

```
// Wrong
```

```
useEffect(() => {
  setTodos(todos.concat(newTodo));
}, []); // Missing 'todos' and 'newTodo' dependencies
```

- Use ESLint's `react-hooks/exhaustive-deps` rule to catch this.

2. Overusing useState:

- For related state, use a single object or `useReducer`:

```
const [state, setState] = useState({ name: '', email: '' });
```

3. Forgetting Cleanup:

- Without cleanup, effects like timers or subscriptions can cause memory leaks:

```
useEffect(() => {
  const id = setInterval(() => console.log('Tick'), 1000);
  return () => clearInterval(id); // Prevents memory leak
}, []);
```

HOOKS: USECONTEXT & USEREDUCER

Hooks allow functional components to manage state and side effects, and `useContext` and `useReducer` are powerful hooks for handling global state and complex state logic, respectively.

- useContext: A hook that provides access to a React Context, allowing components to consume shared data (e.g., theme, user data) without prop drilling. Context is ideal for global state that multiple components need to access.

- useReducer: A hook for managing complex state logic, similar to `useState` but more suitable for state transitions involving multiple actions or interdependent state updates. It uses a reducer function to update state based on dispatched actions, inspired by Redux.

Key Points:

- `useContext` eliminates the need to pass props through multiple component levels.

- `useReducer` is preferred over `useState` when state logic involves multiple actions or complex transitions.

- Combining `useContext` and `useReducer` is a common pattern for managing global state in small to medium-sized apps without external libraries like Redux.

- Both hooks follow the "Rules of Hooks": call them at the top level of functional components or custom hooks, not inside loops or conditions.

Example: Theme Toggle with Todo List

Let’s create an application with a global theme (light/dark mode) managed by `useContext` and a todo list with complex state logic managed by `useReducer`. Users can toggle the theme and manage todos (add, toggle, delete).

1. Create Context (`ThemeContext.jsx`)

```
import { createContext, useContext, useState } from 'react';

// Create Context
const ThemeContext = createContext();

// Theme Provider Component
function ThemeProvider({ children }) {
  const [theme, setTheme] = useState('light');
  const toggleTheme = () => {
    setTheme(prevTheme => (prevTheme === 'light' ? 'dark' : 'light'));
  };
  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}
// Custom hook for accessing theme context
function useTheme() {
  const context = useContext(ThemeContext);
  if (!context) {
    throw new Error('useTheme must be used within a ThemeProvider');
  }
  return context;
}
export { ThemeProvider, useTheme };
```

2. Todo Reducer (`TodoReducer.jsx`)

```
// Reducer function for managing todos
function todoReducer(state, action) {
  switch (action.type) {
    case 'ADD_TODO':
      return [
        ...state,
        {
          id: state.length + 1,
          text: action.payload,
          completed: false,
        },
      ];
    case 'TOGGLE_TODO':
      return state.map(todo =>
        todo.id === action.payload
          ? { ...todo, completed: !todo.completed }
          : todo
      );
    case 'DELETE_TODO':
      return state.filter(todo => todo.id !== action.payload);
    default:
      return state;
  }
}
export default todoReducer;
```

3. Todo List Component (`TodoList.jsx`)

```
import { useReducer } from 'react';
import { useTheme } from './ThemeContext';
import todoReducer from './TodoReducer';
```

```
function TodoList() {
  // Initialize useReducer with the todoReducer and initial state
  const [todos, dispatch] = useReducer(todoReducer, [
    { id: 1, text: 'Learn useContext', completed: false },
    { id: 2, text: 'Learn useReducer', completed: false },
  ]);
  // State for new todo input
  const [newTodo, setNewTodo] = useState('');
  // Access theme context
  const { theme } = useTheme();
  // Event handlers
  const addTodo = () => {
    if (newTodo.trim() === '') return;
    dispatch({ type: 'ADD_TODO', payload: newTodo });
    setNewTodo('');
  };
  const toggleTodo = (id) => {
    dispatch({ type: 'TOGGLE_TODO', payload: id });
  };
  const deleteTodo = (id) => {
    dispatch({ type: 'DELETE_TODO', payload: id });
  };
  return (
    <div
      style={{
        padding: '16px',
        maxWidth: '600px',
        margin: '0 auto',
        background: theme === 'light' ? '#fff' : '#333',
```

```
    color: theme === 'light' ? '#000' : '#fff',
  }}
>
  <h2>Todo List</h2>
  <div>
    <input
      type="text"
      value={newTodo}
      onChange={(e) => setNewTodo(e.target.value)}
      placeholder="Add a new todo"
      style={{ padding: '8px', marginRight: '8px' }}
    />
    <button onClick={addTodo}>Add Todo</button>
  </div>
  <ul style={{ listStyle: 'none', padding: 0 }}>
    {todos.map(todo => (
      <li
        key={todo.id}
        style={{
          textDecoration: todo.completed ? 'line-through' : 'none',
          margin: '8px 0',
          display: 'flex',
          justifyContent: 'space-between',
        }}
      >
        <span
          onClick={() => toggleTodo(todo.id)}
          style={{ cursor: 'pointer' }}
        >
          {todo.text}
```

```
            </span>
            <button
              onClick={() => deleteTodo(todo.id)}
              style={{ color: 'red' }}
            >
              Delete
            </button>
          </li>
        ))}
      </ul>
    </div>
  );
}
export default TodoList;
```

4. Parent Component (`App.jsx`)

```
import { ThemeProvider, useTheme } from './ThemeContext';
import TodoList from './TodoList';
function ThemeToggle() {
  const { theme, toggleTheme } = useTheme();
  return (
    <button
      onClick={toggleTheme}
      style={{ margin: '16px', padding: '8px 16px' }}
    >
      Switch to {theme === 'light' ? 'Dark' : 'Light'} Mode
    </button>
  );
}
```

```
function App() {
  return (
    <ThemeProvider>
      <div>
        <h1>useContext & useReducer Demo</h1>
        <ThemeToggle />
        <TodoList />
      </div>
    </ThemeProvider>
  );
}
export default App;
```

5. Output in Browser

- The app displays a todo list with two initial todos: “Learn useContext” and “Learn useReducer.”

- Users can:

  - Add new todos via the input and “Add Todo” button.

  - Toggle a todo’s completion by clicking its text (strikes through when completed).

  - Delete a todo with the “Delete” button.

  - Toggle between light and dark themes using the “Switch to Dark/Light Mode” button.

- The UI updates reactively based on state changes and theme context.

Key Concepts in the Example

1. useContext:

- The `ThemeContext` provides `theme` and `toggleTheme` to all components within the `ThemeProvider`.

- The `useTheme` custom hook simplifies accessing the context and includes error handling for misuse.

- Used in `TodoList` to style the component based on the current theme and in `ThemeToggle` to switch themes.

2.useReducer:

- The `todoReducer` handles three actions: `ADD_TODO`, `TOGGLE_TODO`, and `DELETE_TODO`.

- `useReducer(todoReducer, initialTodos)` initializes the todo state and provides the `dispatch` function to trigger state updates.

- Actions are dispatched with a `type` and optional `payload` (e.g., `{ type: 'ADD_TODO', payload: 'New Task' }`).

3. Combining Hooks:

- `useContext` manages global theme state, accessible across components.

- `useReducer` manages the todo list's complex state logic locally within `TodoList`.

- `useState` is used for the `newTodo` input, showing how hooks can work together.

Best Practices

1. useContext:

- Use for global state that multiple components need (e.g., theme, user authentication).

- Create a custom hook (e.g., `useTheme`) for cleaner access and to encapsulate context logic.

- Avoid overusing Context for all state; use it only when prop drilling becomes cumbersome.

- Provide meaningful default values or throw errors if the context is used outside a provider.

2. useReducer:

- Use for complex state logic with multiple actions or interdependent state updates.

- Keep the reducer pure (no side effects, predictable output for given input).

- Define action types as constants to avoid typos:

  const ADD_TODO = 'ADD_TODO';

- Combine with `useContext` for global state management in larger apps:

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

  <TodoContext.Provider value={{ state, dispatch }}>

3. General:

- Follow the Rules of Hooks: Call `useContext` and `useReducer` at the top level.

- Use descriptive action types and payloads in `useReducer`.

- Keep Context providers high in the component tree to ensure wide accessibility.

- Optimize performance by memoizing Context values with `useMemo` if they change frequently:

  const value = useMemo(() => ({ theme, toggleTheme }), [theme]);

Insights

- useContext vs. Prop Drilling:

- Use `useContext` to avoid passing props through multiple layers (prop drilling).

- Example: Without Context, you'd need to pass `theme` and `toggleTheme` through every component between `App` and `TodoList`.

- useReducer vs. useState:

- Use `useReducer` when state transitions are complex or involve multiple actions (e.g., todo list with add, toggle, delete).

- Use `useState` for simple, independent state (e.g., form inputs).

- Example: `newTodo` uses `useState` because it's a single value with simple updates.

- Scalability:

- Combine `useContext` and `useReducer` for Redux-like global state management without external libraries.

- For very large apps, consider Redux, Zustand, or Recoil for more robust state management.

- Performance:

- Context updates cause all consumers to re-render. Use `useMemo` or split contexts to minimize re-renders:

  const contextValue = useMemo(() => ({ theme, toggleTheme }), [theme]);

- `useReducer` is efficient for complex state but avoid overcomplicating simple state logic.

Common Pitfalls

1. Using useContext Incorrectly:

- Accessing Context outside a provider causes errors. Always wrap components in the provider or use a custom hook with error handling.

- Example:

  // Wrong

  const { theme } = useContext(ThemeContext); // Error if not in provider

2. Impure Reducers:

- Reducers must be pure (no API calls, no direct state mutations):

```
// Wrong
function reducer(state, action) {
  state.push({ id: 1 }); // Mutates state
  return state;
}
// Correct
function reducer(state, action) {
  return [...state, { id: 1 }];
}
```

3. Overusing Context:

- Don’t use Context for all state; reserve it for truly global data to avoid unnecessary re-renders.

- Example: Local form state should use `useState` or `useReducer`, not Context.

4. Missing Dependencies:

- If `useReducer` dispatch is used in a `useEffect`, ensure it’s not listed as a dependency (it’s stable and doesn’t change):

```
useEffect(() => {
  dispatch({ type: 'INIT' });
}, []); // No need to include 'dispatch'
```

ADVANCED HOOKS: USEREF, USEMEMO, USECALLBACK

React's advanced hooks—`useRef`, `useMemo`, and `useCallback`—are designed to optimize performance, manage mutable references, and handle specific use cases in functional components.

- **useRef:** A hook that creates a mutable reference object that persists across renders. It's commonly used to access DOM elements or store values that don't trigger re-renders when updated.

- **useMemo:** A hook that memoizes a computed value, recomputing it only when its dependencies change. It's used to optimize expensive calculations and prevent unnecessary re-renders.

- **useCallback**: A hook that memoizes a callback function, returning the same function instance unless its dependencies change. It's used to prevent unnecessary re-creations of functions, especially when passing callbacks to child components.

Key Points:

- `useRef` is not just for DOM access; it can store any mutable value that persists without triggering re-renders.

- `useMemo` and `useCallback` are performance optimization tools; use them only when needed to avoid overcomplicating code.

- All three hooks follow the Rules of Hooks: call them at the top level of functional components or custom hooks, not inside loops or conditions.

Example: Optimized Searchable Todo List

Let's create a searchable todo list application that uses:

- `useRef` to focus an input field and track render counts.

- `useMemo` to compute a filtered todo list based on a search query.

- `useCallback` to memoize event handlers passed to child components.

1. Todo Item Component (`TodoItem.jsx`)

```
import { memo } from 'react';

// Memoized to prevent unnecessary re-renders
function TodoItem({ todo, onToggle, onDelete }) {
  return (
    <li
```

```
      style={{
        textDecoration: todo.completed ? 'line-through' : 'none',
        display: 'flex',
        justifyContent: 'space-between',
        margin: '8px 0',
      }}
    >
      <span onClick={() => onToggle(todo.id)} style={{ cursor: 'pointer' }}>
        {todo.text}
      </span>
      <button onClick={() => onDelete(todo.id)} style={{ color: 'red' }}>
        Delete
      </button>
    </li>
  );
}
export default memo(TodoItem); // Memoize to optimize
```

2. Main Todo List Component (`TodoList.jsx`)

```
import { useState, useRef, useMemo, useCallback } from 'react';
import TodoItem from './TodoItem';
function TodoList() {
  // State for todos and search query
  const [todos, setTodos] = useState([
    { id: 1, text: 'Learn useRef', completed: false },
    { id: 2, text: 'Optimize with useMemo', completed: false },
    { id: 3, text: 'Use useCallback', completed: false },
  ]);
  const [searchQuery, setSearchQuery] = useState('');
```

```
// useRef for input focus and render count
const inputRef = useRef(null);
const renderCount = useRef(0);
// Increment render count (doesn't trigger re-render)
renderCount.current += 1;
// useCallback for event handlers
const addTodo = useCallback(() => {
  if (!inputRef.current.value.trim()) return;
  setTodos(prev => [
    ...prev,
    { id: prev.length + 1, text: inputRef.current.value, completed: false },
  ]);
  inputRef.current.value = '';
  inputRef.current.focus();
}, []);
const toggleTodo = useCallback((id) => {
  setTodos(prev =>
    prev.map(todo =>
      todo.id === id ? { ...todo, completed: !todo.completed } : todo
    )
  );
}, []);
const deleteTodo = useCallback((id) => {
  setTodos(prev => prev.filter(todo => todo.id !== id));
}, []);
// useMemo for filtered todos
const filteredTodos = useMemo(() => {
  console.log('Filtering todos...'); // Logs only when dependencies change
  return todos.filter(todo =>
    todo.text.toLowerCase().includes(searchQuery.toLowerCase())
```

```
  );
}, [todos, searchQuery]);
// Focus input on button click
const focusInput = () => {
  inputRef.current.focus();
};
return (
  <div style={{ padding: '16px', maxWidth: '600px', margin: '0 auto' }}>
    <h1>Searchable Todo List</h1>
    <p>Render Count: {renderCount.current}</p>
    <div>
      <input
        ref={inputRef}
        type="text"
        placeholder="Add a new todo"
        onKeyPress={(e) => e.key === 'Enter' && addTodo()}
        style={{ padding: '8px', marginRight: '8px' }}
      />
      <button onClick={addTodo}>Add Todo</button>
      <button onClick={focusInput} style={{ marginLeft: '8px' }}>
        Focus Input
      </button>
    </div>
    <div>
      <input
        type="text"
        value={searchQuery}
        onChange={(e) => setSearchQuery(e.target.value)}
        placeholder="Search todos..."
        style={{ padding: '8px', margin: '8px 0', width: '100%' }}
```

```
        />
      </div>
      <ul style={{ listStyle: 'none', padding: 0 }}>
        {filteredTodos.map(todo => (
          <TodoItem
            key={todo.id}
            todo={todo}
            onToggle={toggleTodo}
            onDelete={deleteTodo}
          />
        ))}
      </ul>
    </div>
  );
}
export default TodoList;
```

3. Parent Component (`App.jsx`)

```
import TodoList from './TodoList';
function App() {
  return (
    <div>
      <h1>Advanced Hooks Demo</h1>
      <TodoList />
    </div>
  );
}
export default App;
```

4. Output in Browser

- Displays a todo list with three initial todos: “Learn useRef,” “Optimize with useMemo,” “Use useCallback.”

- Users can:

  - Add new todos by typing and pressing Enter or clicking “Add Todo.”

  - Toggle a todo’s completion by clicking its text (strikes through when completed).

  - Delete a todo with the “Delete” button.

  - Search todos by typing in the search input (filters in real-time).

  - Click “Focus Input” to focus the todo input field.

- The render count increments with each render, tracked via `useRef`.

- Filtering logs only when `todos` or `searchQuery` changes, thanks to `useMemo`.

- `TodoItem` components don’t re-render unnecessarily due to `useCallback` and `memo`.

Key Concepts in the Example

1. useRef:

- `inputRef` stores a reference to the input DOM element, allowing programmatic focus via `inputRef.current.focus()`.

- `renderCount` tracks the number of renders without triggering re-renders when updated (`renderCount.current += 1`).

- Unlike state, updating a ref’s `.current` property doesn’t cause a re-render.

2. useMemo:

- Memoizes the `filteredTodos` array, recomputing only when `todos` or `searchQuery` changes.

- Prevents expensive filtering operations on every render, improving performance.

- The `console.log` inside `useMemo` demonstrates it runs only when dependencies change.

3. useCallback:

- Memoizes `addTodo`, `toggleTodo`, and `deleteTodo` functions, ensuring they don’t change unless their dependencies change (empty `[]` in this case).

- Combined with `memo` on `TodoItem`, this prevents unnecessary re-renders of `TodoItem` components when the parent re-renders.

Best Practices

1. useRef:

- Use for DOM access (e.g., focusing inputs, measuring elements) or storing mutable values that don't affect rendering (e.g., timers, counters).

- Avoid using refs for values that control rendering; use state instead.

- Example: Use `useRef` for a timer ID, but `useState` for a counter displayed in the UI.

2. useMemo:

- Use for expensive calculations (e.g., filtering large datasets, complex computations).

- Include all dependencies used in the memoized function to avoid stale data.

- Don't overuse; only apply when performance is a concern, as it adds complexity.

- Example:

```
const total = useMemo(() => items.reduce((sum, item) => sum + item.price, 0), [items]);
```

3. useCallback:

- Use when passing callbacks to memoized child components to prevent re-renders.

- Include all dependencies used in the callback to ensure correctness.

- Example:

```
const handleClick = useCallback(() => {
  setCount(prev => prev + value);
}, [value]); // Include 'value' as a dependency
```

- Pair with `React.memo` for maximum effect.

4. General:

- Follow the Rules of Hooks: Call hooks at the top level.

- Use ESLint's `react-hooks/exhaustive-deps` rule to catch missing dependencies in `useMemo` and `useCallback`.

- Profile performance with React Developer Tools to determine if `useMemo` or `useCallback` is necessary.

- Keep components small and focused to make hook usage more manageable.

Insights

- useRef vs. useState:

- Use `useRef` for values that don't affect rendering (e.g., DOM refs, counters for debugging).

- Use `useState` for values that trigger re-renders (e.g., UI data like `todos`).

- Example: `renderCount` uses `useRef` because it's for debugging, not display.

- useMemo vs useCallback:

- `useMemo` memoizes any value (e.g., arrays, objects, computed results).

- `useCallback` is a specialized `useMemo` for functions:

```
// Equivalent to useCallback
const fn = useMemo(() => () => doSomething(), [dep]);
```

- Use `useCallback` for functions passed as props; use `useMemo` for computed values.

- Performance Considerations:

- Overusing `useMemo` or `useCallback` can increase memory usage and code complexity. Only use them when profiling shows a performance issue.

- Example: If `filteredTodos` was a small array, `useMemo` might not be worth the overhead.

- Real-World Use Cases:

- `useRef`: Store timers, track previous state, or access canvas/WebGL contexts.

- `useMemo`: Optimize sorting, filtering, or heavy computations in data-heavy apps.

- `useCallback`: Stabilize callbacks in forms, lists, or components wrapped in `React.memo`.

- Combining with Other Hooks:

- Use `useRef` in `useEffect` to store values that persist across renders:

```
useEffect(() => {
  const timer = setInterval(() => console.log('Tick'), 1000);
  ref.current = timer;
  return () => clearInterval(ref.current);
}, []);
```

- Use `useMemo` to optimize Context values:

```
const value = useMemo(() => ({ state, dispatch }), [state]);
```

Common Pitfalls

1. Overusing useMemo/useCallback:

- Adding `useMemo` or `useCallback` unnecessarily can make code harder to maintain without significant performance gains.

- Profile with React Developer Tools to justify their use.

2. Missing Dependencies:

- Omitting dependencies in `useMemo` or `useCallback` can lead to stale data or bugs:

// Wrong

play

const filtered = useMemo(() => todos.filter(t => t.text.includes(search)), [todos]); // Missing 'search' dependency

- Use ESLint to catch missing dependencies.

3. Mutating Refs Incorrectly:

- `useRef` updates don't trigger re-renders, so don't use refs for UI-related data:

// Wrong

const count = useRef(0);

count.current += 1; // Won't update UI

4. Unnecessary Re-renders:

- Without `useCallback`, child components like `TodoItem` may re-render unnecessarily if passed new function instances:

// Without useCallback

const onToggle = () => setTodos(...); // New instance every render

HANDLING FORMS & EVENTS IN REACT

In React, forms and events are managed in a controlled, declarative way, leveraging React's state management and event-handling system. Forms in React typically use controlled components, where form inputs are bound to state, ensuring that React is the single source of truth. Events, such as clicks, input changes, or submissions, are handled using React's synthetic event system, which wraps native browser events for consistency and performance.

Key Concepts

1. Controlled Components:

- A controlled component is an input whose value is controlled by React state.

- The `value` attribute of the input is tied to a state variable, and the `onChange` event updates the state.

- This ensures predictable behavior and makes it easier to validate or manipulate input data.

2. Event Handling:

- React uses synthetic events, a cross-browser wrapper around native DOM events (e.g., `click`, `change`, `submit`).

- Event handlers are passed as props (e.g., `onClick`, `onChange`) and are typically defined as arrow functions or methods in functional components.

3. Form Submission:

- The `<form>` element's `onSubmit` event is used to handle form submissions.

- You prevent the default browser behavior (page refresh) using `event.preventDefault()`.

4. Common Form Elements:

- Inputs (`<input>`, `<textarea>`, `<select>`), checkboxes, and radio buttons are commonly used in forms.

- Each has specific event-handling patterns (e.g., `checked` for checkboxes instead of `value`).

5. Validation & Feedback:

- Form validation can be handled by updating state based on input values and displaying error messages.

- `useEffect` can be used for real-time validation or side effects after input changes.

Example 1: Basic Controlled Form

Let's create a simple login form with controlled inputs for email and password, including form submission.

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

function LoginForm() {
  // State for form inputs
  const [formData, setFormData] = useState({
    email: '',
    password: '',
  });
  // Handle input changes
  const handleChange = (e) => {
    const { name, value } = e.target; // Destructure name and value from input
    setFormData((prev) => ({
```

```
    ...prev,
    [name]: value, // Dynamically update state based on input name
  }));
};
// Handle form submission
const handleSubmit = (e) => {
  e.preventDefault(); // Prevent page refresh
  console.log('Form submitted:', formData);
  // Example: Call an API or perform validation here
};
return (
  <form onSubmit={handleSubmit}>
    <div>
      <label htmlFor="email">Email:</label>
      <input
        type="email"
        id="email"
        name="email"
        value={formData.email}
        onChange={handleChange}
        required
      />
    </div>
    <div>
      <label htmlFor="password">Password:</label>
      <input
        type="password"
        id="password"
        name="password"
        value={formData.password}
```

```
        onChange={handleChange}
        required
       />
     </div>
     <button type="submit">Login</button>
   </form>
  );
}
export default LoginForm;
```

Explanation:

- State Management: `useState` holds the form data in an object (`formData`) with keys for `email` and `password`.

- Controlled Inputs: Each `<input>` has a `value` tied to `formData` and an `onChange` handler to update state.

- Dynamic Updates: The `handleChange` function uses the input's `name` attribute to dynamically update the corresponding state field.

- Form Submission: The `handleSubmit` function prevents the default form submission behavior and logs the form data (in a real app, you'd likely send this to an API).

- JSX: The form is written in JSX, with `htmlFor` for labels (React's equivalent of HTML's `for` attribute).

Example 2: Form with Validation and Checkboxes

Let's extend the concept to include validation and a checkbox for a "terms of service" agreement.

```
import React, { useState, useEffect } from 'react';
function RegistrationForm() {
  const [formData, setFormData] = useState({
    username: '',
    agreeToTerms: false,
  });
  const [errors, setErrors] = useState({});
```

```
// Validate form in real-time using useEffect
useEffect(() => {
  const newErrors = {};
  if (formData.username.length < 3) {
    newErrors.username = 'Username must be at least 3 characters';
  }
  if (!formData.agreeToTerms) {
    newErrors.agreeToTerms = 'You must agree to the terms';
  }
  setErrors(newErrors);
}, [formData]); // Run when formData changes
const handleChange = (e) => {
  const { name, value, type, checked } = e.target;
  setFormData((prev) => ({
    ...prev,
    [name]: type === 'checkbox' ? checked : value,
  }));
};
const handleSubmit = (e) => {
  e.preventDefault();
  if (Object.keys(errors).length === 0) {
    console.log('Form submitted successfully:', formData);
  } else {
    console.log('Form has errors:', errors);
  }
};
return (
  <form onSubmit={handleSubmit}>
    <div>
```

```
        <label htmlFor="username">Username:</label>
        <input
          type="text"
          id="username"
          name="username"
          value={formData.username}
          onChange={handleChange}
        />
        {errors.username && <span style={{ color: 'red' }}>{errors.username}</span>}
      </div>
      <div>
        <label>
          <input
            type="checkbox"
            name="agreeToTerms"
            checked={formData.agreeToTerms}
            onChange={handleChange}
          />
          I agree to the terms
        </label>
        {errors.agreeToTerms && <span style={{ color: 'red' }}>{errors.agreeToTerms}</span>}
      </div>
      <button type="submit" disabled={Object.keys(errors).length > 0}>
        Register
      </button>
    </form>
  );
}
export default RegistrationForm;
```

Explanation:

- Checkbox Handling: The checkbox uses the `checked` attribute instead of `value`, and `handleChange` checks the input `type` to handle `checked` for checkboxes.

- Validation with useEffect: The `useEffect` hook runs whenever `formData` changes, validating the username length and terms agreement. Errors are stored in the `errors` state.

- Error Display: Errors are conditionally rendered below each input using JSX.

- Submit Button: The button is disabled if there are validation errors, ensuring the form can't be submitted until valid.

- Dynamic State Updates: The `handleChange` function handles both text inputs and checkboxes dynamically.

Example 3: Select Dropdown and Multiple Inputs

This example shows a form with a `<select>` dropdown and multiple inputs, demonstrating how to handle different input types.

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

function ProfileForm() {
  const [formData, setFormData] = useState({
    name: '',
    role: 'developer',
  });
  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData((prev) => ({
      ...prev,
      [name]: value,
    }));
  };
  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Profile created:', formData);
  };
```

```
  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label htmlFor="name">Name:</label>
        <input
          type="text"
          id="name"
          name="name"
          value={formData.name}
          onChange={handleChange}
        />
      </div>
      <div>
        <label htmlFor="role">Role:</label>
        <select id="role" name="role" value={formData.role} onChange={handleChange}>
          <option value="developer">Developer</option>
          <option value="designer">Designer</option>
          <option value="manager">Manager</option>
        </select>
      </div>
      <button type="submit">Save Profile</button>
    </form>
  );
}
export default ProfileForm;
```

Explanation:

- Select Dropdown: The `<select>` element is controlled like an `<input>`, with `value` tied to state and `onChange` updating it.

- Reusable Handler: The `handleChange` function works for both `<input>` and `<select>` by using the `name` attribute.

- Form Submission: The `handleSubmit` function logs the form data, but you could extend it to send data to an API or update a context (using `useContext` from your studied topics).

Best Practices for Forms & Events

1. Use Controlled Components: Always tie form inputs to state for predictable behavior and easier validation.

2. Centralize State: Store all form data in a single state object (as shown in examples) to simplify updates and submission.

3. Prevent Default Behaviour: Always call `e.preventDefault()` in `onSubmit` to avoid page refreshes.

4. Validate Early: Use `useEffect` or inline validation to provide real-time feedback to users.

5. Optimize Performance: For complex forms, consider `useMemo` or `useCallback` (from your studied topics) to memoize expensive validation logic or event handlers.

```
const handleChange = useCallback((e) => {
  const { name, value } = e.target;
  setFormData((prev) => ({ ...prev, [name]: value }));
}, []); // Empty deps since setFormData is stable
```

6. Accessibility: Use `htmlFor` with labels and ensure inputs have proper attributes like `id`, `name`, and `required`.

7. Error Handling: Display clear error messages and disable submission when the form is invalid.

8. Form Libraries: For complex forms, consider libraries like Formik or React Hook Form, which simplify validation and state management (beyond the scope of this explanation but good to know).

Common Pitfalls

- Forgetting e.preventDefault(): Leads to unexpected page refreshes on form submission.

- Uncontrolled Components: Avoid setting only `defaultValue` without state, as it makes forms harder to manage.

- Overcomplicating Event Handlers: Keep handlers simple and reusable (e.g., use a single `handleChange` for multiple inputs).

- Ignoring Accessibility: Always pair labels with inputs and test for keyboard navigation.

Connecting to Studied Topics

- JSX: Forms are built with JSX elements like `<form>`, `<input>`, and `<select>`.

- useState: Manages form data and validation errors.

- useEffect: Useful for side effects like real-time validation or API calls after submission.

- useRef: Can be used for uncontrolled forms or to focus inputs programmatically (e.g., `const inputRef = useRef(); inputRef.current.focus();`).

- useContext: Useful for sharing form data across components (e.g., a multi-step form wizard).

- useReducer: For complex forms with many fields or interdependent state, `useReducer` can replace multiple `useState` calls.

```
const initialState = { email: '', password: '' };
const reducer = (state, action) => {
  switch (action.type) {
    case 'UPDATE_FIELD':
      return { ...state, [action.field]: action.value };
    default:
      return state;
  }
};
const [state, dispatch] = useReducer(reducer, initialState);
```

Advanced Example: Multi-Step Form with Context

If you're ready to combine forms with `useContext` and `useReducer`, here's a brief example of a multi-step form.

```
import React, { useContext, useReducer } from 'react';
import { createContext } from 'react';
// Form context
const FormContext = createContext();
// Reducer for form state
const formReducer = (state, action) => {
  switch (action.type) {
    case 'UPDATE_FIELD':
```

```
      return { ...state, [action.field]: action.value };
    default:
      return state;
  }
};
function MultiStepForm() {
  const [formData, dispatch] = useReducer(formReducer, { name: '', email: '' });
  const [step, setStep] = useState(1);
  const nextStep = () => setStep((prev) => prev + 1);
  const prevStep = () => setStep((prev) => prev - 1);
  return (
    <FormContext.Provider value={{ formData, dispatch }}>
      {step === 1 && <Step1 />}
      {step === 2 && <Step2 />}
      <button onClick={prevStep} disabled={step === 1}>Previous</button>
      <button onClick={nextStep} disabled={step === 2}>Next</button>
    </FormContext.Provider>
  );
}
function Step1() {
  const { formData, dispatch } = useContext(FormContext);
  return (
    <div>
      <label>Name:</label>
      <input
        value={formData.name}
        onChange={(e) =>
          dispatch({ type: 'UPDATE_FIELD', field: 'name', value: e.target.value })
        }
      />
```

```
    </div>
  );
}
function Step2() {
  const { formData, dispatch } = useContext(FormContext);
  return (
    <div>
      <label>Email:</label>
      <input
        value={formData.email}
        onChange={(e) =>
          dispatch({ type: 'UPDATE_FIELD', field: 'email', value: e.target.value })
        }
      />
    </div>
  );
}
export default MultiStepForm;
```

Explanation:

- useContext: Shares form state across steps without prop drilling.

- useReducer: Manages form data in a centralized way, ideal for complex forms.

- Event Handling: Input changes dispatch actions to update the reducer state.

ROUTING WITH REACT ROUTER

React Router is the de facto library for handling client-side routing in React applications. It allows you to create single-page applications (SPAs) where the UI updates dynamically as users navigate between "pages" without full page reloads. React Router synchronizes the URL with your application's UI, enabling navigation, dynamic routes, and nested layouts.

Key Concepts

1. Core Components:

- `<BrowserRouter>`: Wraps the app to enable routing using the HTML5 History API.
- `<Routes>` and `<Route>`: Define the mapping between URLs and components.
- `<Link>` and `<NavLink>`: Create navigation links that update the URL without reloading.
- `<Outlet>`: Renders child routes in nested routing setups.

2. Dynamic Routing:

- Routes can include parameters (e.g., `/users/:id`) to render dynamic content.
- Hooks like `useParams` and `useNavigate` provide access to route parameters and programmatic navigation.

3. Nested Routes:

- Routes can be nested to create layouts where parts of the UI (e.g., a sidebar) remain constant while others change.

4. Programmatic Navigation:

- The `useNavigate` hook allows navigation via code (e.g., after form submission).

5. Route Protection:

- Protect routes (e.g., requiring authentication) using conditional rendering or custom route components.

Setup

To use React Router, install it via npm:

npm install react-router-dom

Ensure your project is set up with React. We'll use the latest version of React Router (v6) for these examples, as it's the standard in 2025.

Example 1: Basic Routing

This example demonstrates a simple app with multiple pages and navigation using React Router.

```
import React from 'react';
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';
// Home component
function Home() {
  return <h2>Home Page</h2>;
}
// About component
function About() {
  return <h2>About Page</h2>;
}
// App component with routing
function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link> | <Link to="/about">About</Link>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
}
export default App;
```

Explanation:

- BrowserRouter: Wraps the app to enable routing.

- Link: Creates clickable links that update the URL without reloading.

- Routes and Route: Map URLs (`path`) to components (`element`).

- JSX: The `<Link>` and `<Route>` components are written in JSX, leveraging your knowledge of JSX fundamentals.

When you click the "Home" link, the URL becomes `/`, and the `Home` component renders. Clicking "About" changes the URL to `/about` and renders the `About` component.

Example 2: Dynamic Routing with Parameters

This example shows how to handle dynamic routes (e.g., `/users/:id`) and use the `useParams` hook.

```
import React from 'react';

import { BrowserRouter, Routes, Route, Link, useParams } from 'react-router-dom';

// User component with dynamic parameter

function User() {

  const { id } = useParams(); // Get the :id parameter from the URL

  return <h2>User ID: {id}</h2>;

}

function App() {

  return (

    <BrowserRouter>

      <nav>

        <Link to="/">Home</Link> | <Link to="/user/1">User 1</Link> | <Link to="/user/2">User 2</Link>

      </nav>

      <Routes>

        <Route path="/" element={<h2>Home Page</h2>} />

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

      </Routes>

    </BrowserRouter>

  );

}
```

```
export default App;
```

Explanation:

- Dynamic Route: The `path="/user/:id"` defines a route with a dynamic `id` parameter.

- useParams: The `useParams` hook extracts the `id` from the URL (e.g., `/user/123` gives `{ id: "123" }`).

- Navigation: Clicking the links updates the URL and renders the `User` component with the corresponding `id`.

This builds on your knowledge of props, as `useParams` provides route parameters similar to how props pass data to components.

Example 3: Programmatic Navigation with Forms

Combining routing with Handling Forms & Events, this example navigates programmatically after a form submission using `useNavigate`.

```
import React, { useState } from 'react';
import { BrowserRouter, Routes, Route, Link, useNavigate } from 'react-router-dom';
// Login form component
function Login() {
  const [formData, setFormData] = useState({ username: '' });
  const navigate = useNavigate(); // Hook for programmatic navigation
  const handleChange = (e) => {
    setFormData({ username: e.target.value });
  };
  const handleSubmit = (e) => {
    e.preventDefault();
    // Simulate login success
    console.log('Logged in:', formData);
    navigate('/dashboard'); // Navigate to dashboard after submission
  };
  return (
    <form onSubmit={handleSubmit}>
      <label>
```

```
        Username:
        <input
          type="text"
          value={formData.username}
          onChange={handleChange}
        />
      </label>
      <button type="submit">Login</button>
    </form>
  );
}
function Dashboard() {
  return <h2>Welcome to the Dashboard!</h2>;
}
function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link> | <Link to="/login">Login</Link>
      </nav>
      <Routes>
        <Route path="/" element={<h2>Home Page</h2>} />
        <Route path="/login" element={<Login />} />
        <Route path="/dashboard" element={<Dashboard />} />
      </Routes>
    </BrowserRouter>
  );
}
export default App;
```

Explanation:

- useNavigate: The `useNavigate` hook provides a function to programmatically change routes (e.g., `navigate('/dashboard')`).

- Form Integration: The form uses controlled inputs (from your Forms & Events knowledge) and triggers navigation on submission.

- useState: Manages the form's state, showing how routing integrates with state management.

This example ties directly to your understanding of useState for form state and event handling for form submission.

Example 4: Nested Routes and Layouts

Nested routes allow you to create layouts where parts of the UI remain constant. This example uses `<Outlet>` for nested routes.

```
import React from 'react';
import { BrowserRouter, Routes, Route, Link, Outlet } from 'react-router-dom';
// Layout component with nested routes
function Layout() {
  return (
    <div>
      <nav>
        <Link to="/settings/profile">Profile</Link> | <Link to="/settings/account">Account</Link>
      </nav>
      <h2>Settings</h2>
      <Outlet /> {/* Renders child routes */}
    </div>
  );
}
function Profile() {
  return <h3>Profile Settings</h3>;
}
function Account() {
  return <h3>Account Settings</h3>;}
```

```
function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<h2>Home Page</h2>} />
        <Route path="/settings" element={<Layout />}>
          <Route path="profile" element={<Profile />} />
          <Route path="account" element={<Account />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}
export default App;
```

Explanation:

- Nested Routes: The `<Route>` for `/settings` includes child routes (`profile`, `account`) rendered via `<Outlet>`.

- Layout: The `Layout` component provides a shared UI (e.g., navigation) for all `/settings/*` routes.

- URLs: `/settings/profile` renders the `Layout` with `Profile` inside the `<Outlet>`, and `/settings/account` renders `Account`.

This leverages your knowledge of components and JSX to structure complex UIs.

Example 5: Protected Routes

Protected routes restrict access (e.g., requiring authentication). This example uses a custom component to protect the dashboard.

```
import React, { useState } from 'react';
import { BrowserRouter, Routes, Route, Link, Navigate } from 'react-router-dom';
// Protected route component
function ProtectedRoute({ isAuthenticated, children }) {
  return isAuthenticated ? children : <Navigate to="/login" />;}
```

```
function Login() {
  const [isAuthenticated, setIsAuthenticated] = useState(false);
  const navigate = useNavigate();
  const handleLogin = () => {
    setIsAuthenticated(true);
    navigate('/dashboard');
  };
  return (
    <div>
      <h2>Login</h2>
      <button onClick={handleLogin}>Login</button>
    </div>
  );
}
function Dashboard() {
  return <h2>Dashboard (Protected)</h2>;
}
function App() {
  const [isAuthenticated, setIsAuthenticated] = useState(false); // Global auth state
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link> | <Link to="/login">Login</Link> | <Link to="/dashboard">Dashboard</Link>
      </nav>
      <Routes>
        <Route path="/" element={<h2>Home Page</h2>} />
        <Route path="/login" element={<Login />} />
        <Route
          path="/dashboard"
          element={
```

```
          <ProtectedRoute isAuthenticated={isAuthenticated}>
            <Dashboard />
          </ProtectedRoute>
        }
      />
    </Routes>
  </BrowserRouter>
 );
}
export default App;
```

Explanation:

- ProtectedRoute: A custom component that conditionally renders its `children` or redirects using `<Navigate>`.

- useState: Manages authentication state (in a real app, this might come from `useContext` or an API).

- Navigate: Programmatically redirects to `/login` if the user isn't authenticated.

- useContext: You could enhance this by storing `isAuthenticated` in a context for global access (from your useContext knowledge).

Best Practices for React Router

1. Use BrowserRouter: Always wrap your app in `<BrowserRouter>` for client-side routing.

2. Leverage Hooks: Use `useNavigate`, `useParams`, and `useLocation` for dynamic navigation and route data.

3. Organize Routes: Group related routes under a parent route with `<Outlet>` for layouts.

4. Optimize Performance: Use `useCallback` for event handlers passed to `<Link>` or navigation functions to prevent unnecessary re-renders.

```
const handleNavigate = useCallback(() => navigate('/dashboard'), [navigate]);
```

5. Handle 404s: Add a catch-all route (`<Route path="*" element={<NotFound />} />`) for invalid URLs.

6. Accessibility: Ensure `<Link>` components have descriptive text and are keyboard-navigable.

7. Protect Routes: Use protected route components or context for authentication logic.

8. Combine with Forms: Use programmatic navigation after form submissions (as shown in Example 3).

Common Pitfalls

- Not Using `<Link>`: Avoid `<a>` tags, as they cause full page reloads. Use `<Link>` or `useNavigate`.

- Ignoring Nested Routes: Overcomplicating layouts by not using `<Outlet>` for nested routes.

- Uncontrolled Navigation: Forgetting to handle redirects for unauthorized users.

- Outdated Syntax: Ensure you're using React Router v6 syntax (e.g., `element` instead of `component` in `<Route>`).

Connecting to Studied Topics

- JSX: React Router components (`<Route>`, `<Link>`, `<Outlet>`) are JSX elements.

- Components: Routes render functional components, leveraging your component knowledge.

- Props: Route parameters and navigation props are passed to components (similar to regular props).

- useState/useEffect: Manage route-specific state or perform side effects (e.g., fetching data with `useEffect` when a route changes).

```
useEffect(() => {
  // Fetch data when route changes
  fetch(`/api/user/${id}`).then((res) => res.json()).then(setData);
}, [id]); // Depends on useParams
```

- useContext: Share authentication state or app-wide data across routes.

- useReducer: Manage complex navigation state (e.g., wizard-like multi-step forms).

- useRef: Focus elements on route changes (e.g., `useRef` to focus an input on a new page).

- useMemo/useCallback: Optimize performance for navigation handlers or computed route data.

Advanced Example: Multi-Step Form with Routing

Building on your Forms & Events knowledge, here's a multi-step form using React Router for navigation, combining `useContext` and `useReducer`.

```
import React, { useContext, useReducer } from 'react';
```

```
import { BrowserRouter, Routes, Route, Link, useNavigate } from 'react-router-dom';
import { createContext } from 'react';
// Form context
const FormContext = createContext();
// Reducer for form state
const formReducer = (state, action) => {
  switch (action.type) {
    case 'UPDATE_FIELD':
      return { ...state, [action.field]: action.value };
    default:
      return state;
  }
};
// Step 1: Name
function Step1() {
  const { formData, dispatch } = useContext(FormContext);
  const navigate = useNavigate();
  const handleSubmit = (e) => {
    e.preventDefault();
    navigate('/form/step2');
  };
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input
          value={formData.name}
          onChange={(e) =>
            dispatch({ type: 'UPDATE_FIELD', field: 'name', value: e.target.value })
          }
```

```
        />
      </label>
      <button type="submit">Next</button>
    </form>
  );
}
// Step 2: Email
function Step2() {
  const { formData, dispatch } = useContext(FormContext);
  const navigate = useNavigate();
  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Form submitted:', formData);
    navigate('/form/complete');
  };
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Email:
        <input
          value={formData.email}
          onChange={(e) =>
            dispatch({ type: 'UPDATE_FIELD', field: 'email', value: e.target.value })
          }
        />
      </label>
      <button type="submit">Submit</button>
    </form>
  );}
function Complete() {
```

```
  return <h2>Form Completed!</h2>;
}
function App() {
  const [formData, dispatch] = useReducer(formReducer, { name: '', email: '' });
  return (
    <BrowserRouter>
      <FormContext.Provider value={{ formData, dispatch }}>
        <nav>
          <Link to="/form/step1">Step 1</Link> | <Link to="/form/step2">Step 2</Link>
        </nav>
        <Routes>
          <Route path="/form/step1" element={<Step1 />} />
          <Route path="/form/step2" element={<Step2 />} />
          <Route path="/form/complete" element={<Complete />} />
        </Routes>
      </FormContext.Provider>
    </BrowserRouter>
  );
}
export default App;
```

Explanation:

- useContext/useReducer: Centralizes form state across routes, leveraging your knowledge of these hooks.

- useNavigate: Moves between form steps after submission.

- Routing: Each form step is a separate route, making navigation intuitive.

- Forms & Events: Each step uses controlled inputs and form submission, tying to your previous topic.

STATE MANAGEMENT WITH THE CONTEXT API

The Context API is a built-in React feature for managing global state, allowing you to share data across components without prop drilling (passing props through multiple layers of components). It's ideal for state that needs to be accessed by many components, such as user authentication, theme settings, or form data in a multi-step form. The Context API is lightweight and works well with hooks like `useContext` and `useReducer` (which you've studied) for scalable state management.

Key Concepts

1. Creating a Context:

- Use `React.createContext()` to create a context object.

- The context provides a `Provider` to supply data and a `Consumer` (or `useContext` hook) to access it.

2. Provider:

- The `<Context.Provider>` component wraps part or all of your app, making the context value available to descendants.

- The `value` prop of the provider holds the shared state.

3. Consuming Context:

- Use the `useContext` hook in functional components to access the context value.

- Alternatively, use `<Context.Consumer>` for class components or complex rendering (less common with hooks).

4. Combining with useReducer:

- For complex state logic, pair the Context API with `useReducer` to manage global state in a predictable way.

5. When to Use Context:

- Use for global state like user authentication, theme, or app-wide settings.

- Avoid for local state that can be managed with `useState` in a single component.

Example 1: Basic Context for Theme Toggle

This example shows a simple theme toggle using the Context API to share the theme state across components.

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

// Create a Theme Context

const ThemeContext = createContext();
```

```
// App component with Theme Provider
function App() {
  const [theme, setTheme] = useState('light');
  const toggleTheme = () => {
    setTheme((prev) => (prev === 'light' ? 'dark' : 'light'));
  };
  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      <div style={{ background: theme === 'light' ? '#fff' : '#333', color: theme === 'light' ? '#000'
: '#fff' }}>
        <Header />
        <Content />
      </div>
    </ThemeContext.Provider>
  );
}
// Header component consuming context
function Header() {
  const { theme, toggleTheme } = useContext(ThemeContext);
  return (
    <header>
      <h1>My App</h1>
      <button onClick={toggleTheme}>
        Switch to {theme === 'light' ? 'Dark' : 'Light'} Theme
      </button>
    </header>
  );
}
// Content component consuming context
function Content() {
  const { theme } = useContext(ThemeContext);
```

```
  return <p>This is the content with {theme} theme.</p>;
}
export default App;
```

Explanation:

- createContext: Creates `ThemeContext` to hold the theme state and toggle function.

- Provider: Wraps the app, providing `theme` and `toggleTheme` to all descendants.

- useContext: The `Header` and `Content` components use `useContext(ThemeContext)` to access the theme data.

- useState: Manages the theme state locally in `App`, which is shared via context.

- JSX & Events: The button in `Header` uses an `onClick` event (from Forms & Events) to toggle the theme.

This example avoids prop drilling, as `Header` and `Content` access the theme directly without passing props through intermediate components.

Example 2: Context with useReducer for User Authentication

This example combines the Context API with `useReducer` to manage global authentication state, integrating with React Router for protected routes.

```
import React, { createContext, useContext, useReducer } from 'react';
import { BrowserRouter, Routes, Route, Link, Navigate } from 'react-router-dom';
// Create Auth Context
const AuthContext = createContext();
// Reducer for auth state
const authReducer = (state, action) => {
  switch (action.type) {
    case 'LOGIN':
      return { ...state, isAuthenticated: true, user: action.payload };
    case 'LOGOUT':
      return { ...state, isAuthenticated: false, user: null };
    default:
      return state;
  }
};
```

```
// Protected Route component
function ProtectedRoute({ children }) {
  const { state } = useContext(AuthContext);
  return state.isAuthenticated ? children : <Navigate to="/login" />;
}
// Login component
function Login() {
  const { dispatch } = useContext(AuthContext);
  const [username, setUsername] = useState('');
  const handleSubmit = (e) => {
    e.preventDefault();
    dispatch({ type: 'LOGIN', payload: { username } });
  };
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Username:
        <input value={username} onChange={(e) => setUsername(e.target.value)} />
      </label>
      <button type="submit">Login</button>
    </form>
  );
}
// Dashboard component
function Dashboard() {
  const { state, dispatch } = useContext(AuthContext);
  return (
    <div>
      <h2>Welcome, {state.user?.username}!</h2>
      <button onClick={() => dispatch({ type: 'LOGOUT' })}>Logout</button>
```

```
    </div>
  );
}
// App component
function App() {
  const [state, dispatch] = useReducer(authReducer, {
    isAuthenticated: false,
    user: null,
  });
  return (
    <AuthContext.Provider value={{ state, dispatch }}>
      <BrowserRouter>
        <nav>
          <Link to="/">Home</Link> | <Link to="/login">Login</Link> | <Link to="/dashboard">Dashboard</Link>
        </nav>
        <Routes>
          <Route path="/" element={<h2>Home Page</h2>} />
          <Route path="/login" element={<Login />} />
          <Route
            path="/dashboard"
            element={
              <ProtectedRoute>
                <Dashboard />
              </ProtectedRoute>
            }
          />
        </Routes>
      </BrowserRouter>
    </AuthContext.Provider>
  );}
```

export default App;

Explanation:

- useReducer: Manages authentication state (`isAuthenticated`, `user`) with actions for login and logout.

- Context Provider: Wraps the app, providing `state` and `dispatch` to all components.

- useContext: The `Login`, `Dashboard`, and `ProtectedRoute` components consume the context to access or modify auth state.

- React Router: Integrates with routing to protect the `/dashboard` route and redirect unauthenticated users to `/login`.

- Forms & Events: The `Login` component uses a controlled form (from your Forms & Events knowledge) to dispatch a login action.

- useState: Used locally in `Login` for form input state, showing how local and global state coexist.

This example demonstrates how the Context API scales for app-wide state management, especially when paired with `useReducer`.

Example 3: Multi-Step Form with Context and Routing

Building on your knowledge of Forms & Events and React Router, this example revisits the multi-step form, using the Context API to share form data across routes.

```
import React, { createContext, useContext, useReducer } from 'react';
import { BrowserRouter, Routes, Route, Link, useNavigate } from 'react-router-dom';
// Create Form Context
const FormContext = createContext();
// Reducer for form state
const formReducer = (state, action) => {
  switch (action.type) {
    case 'UPDATE_FIELD':
      return { ...state, [action.field]: action.value };
    default:
      return state;
  }
};
```

```
// Step 1: Name
function Step1() {
  const { state, dispatch } = useContext(FormContext);
  const navigate = useNavigate();
  const handleSubmit = (e) => {
    e.preventDefault();
    navigate('/form/step2');
  };
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input
          value={state.name}
          onChange={(e) =>
            dispatch({ type: 'UPDATE_FIELD', field: 'name', value: e.target.value })
          }
        />
      </label>
      <button type="submit">Next</button>
    </form>
  );
}
// Step 2: Email
function Step2() {
  const { state, dispatch } = useContext(FormContext);
  const navigate = useNavigate();
  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Form submitted:', state);
```

```
    navigate('/form/complete');
  };
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Email:
        <input
          value={state.email}
          onChange={(e) =>
            dispatch({ type: 'UPDATE_FIELD', field: 'email', value: e.target.value })
          }
        />
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}
// Completion page
function Complete() {
  const { state } = useContext(FormContext);
  return (
    <div>
      <h2>Form Completed!</h2>
      <p>Name: {state.name}</p>
      <p>Email: {state.email}</p>
    </div>
  );
}
```

```
// App component
function App() {
  const [state, dispatch] = useReducer(formReducer, { name: '', email: '' });
  return (
    <FormContext.Provider value={{ state, dispatch }}>
      <BrowserRouter>
        <nav>
          <Link to="/form/step1">Step 1</Link> | <Link to="/form/step2">Step 2</Link>
        </nav>
        <Routes>
          <Route path="/form/step1" element={<Step1 />} />
          <Route path="/form/step2" element={<Step2 />} />
          <Route path="/form/complete" element={<Complete />} />
        </Routes>
      </BrowserRouter>
    </FormContext.Provider>
  );
}
export default App;
```

Explanation:

- Context API: Shares form data (`name`, `email`) across routes without prop drilling.

- useReducer: Manages form state centrally, making it easy to update fields from any step.

- React Router: Each form step is a separate route, and `useNavigate` handles navigation.

- Forms & Events: Controlled inputs and form submissions (from Forms & Events) integrate with context and routing.

- useContext: Each step consumes the context to access and update form data.

This example ties together useContext, useReducer, Forms & Events, and React Router, showing how the Context API unifies state across a complex app.

Best Practices for Context API

1. Use Sparingly: Only use Context for truly global state (e.g., auth, theme, or app-wide settings). Local state should stay in `useState`.

2. Combine with useReducer: For complex state logic, pair Context with `useReducer` instead of multiple `useState` calls.

3. Organize Contexts: Create separate contexts for unrelated concerns (e.g., `AuthContext`, `ThemeContext`) to avoid bloated providers.

4. Optimize Performance: Use `useMemo` or `useCallback` to prevent unnecessary re-renders when passing objects or functions in the context value.

```jsx:disable-run
const value = useMemo(() => ({ state, dispatch }), [state]);
<AuthContext.Provider value={value}>
```

5. Type Safety: Use PropTypes or TypeScript to define the shape of context values.

6. Accessibility: Ensure UI elements affected by context (e.g., theme changes) are accessible (e.g., sufficient contrast).

7. Testing: Test context consumers with tools like `@testing-library/react` to ensure components react to state changes.

Common Pitfalls

- Overusing Context: Don’t use Context for state that’s only needed by a few components—use props or local state instead.

- Re-rendering Issues: Without `useMemo` or `useCallback`, context updates can cause unnecessary re-renders of all consumers.

```
// Bad: New object on every render
<ThemeContext.Provider value={{ theme, toggleTheme }}>
// Good: Memoized value
const value = useMemo(() => ({ theme, toggleTheme }), [theme]);
<ThemeContext.Provider value={value}>
```

- Nested Providers: Too many nested providers can make the app hard to maintain. Keep the provider hierarchy shallow.

- Ignoring Initial State: Always provide a default value for `createContext` to avoid errors when a component is used outside a provider.

Connecting to Studied Topics

- useContext: The primary way to consume context in functional components, which you've already studied.

- useReducer: Pairs with Context for complex state management, as shown in the auth and form examples.

- useState: Used for simpler context state (e.g., theme toggle).

- Forms & Events: Context is ideal for sharing form data across components or routes.

- React Router: Context ensures consistent state across routes (e.g., auth state for protected routes).

- useRef: Can store context-related data that doesn't trigger re-renders (e.g., a ref to track the previous context value).

- useMemo/useCallback: Optimize context values to prevent unnecessary re-renders.

```
const toggleTheme = useCallback(() => {
  setTheme((prev) => (prev === 'light' ? 'dark' : 'light'));
}, []);
```

When to Use Context vs. Other State Management

- Context API: Best for lightweight global state (e.g., auth, theme, small forms). No external dependencies.

- Redux: Use for large-scale apps with complex state and debugging needs (e.g., time-travel debugging).

- Zustand/MobX: Lightweight alternatives to Redux for more flexibility than Context.

- useState/useReducer: Use for local component state unless it needs to be shared globally.

STATE MANAGEMENT WITH REDUX

What is Redux?

Redux is a predictable state management library for JavaScript applications, commonly used with React to manage complex, shared state across components. Unlike the Context API, which is built into React and primarily handles state sharing, Redux provides a centralized, predictable, and scalable approach to state management. It's particularly useful for large applications where state needs to be accessed or modified by many components, or when state changes need to be tracked and debugged systematically.

Redux follows three core principles:

1. Single Source of Truth: The entire application state is stored in a single JavaScript object called the store.

2. State is Read-Only: The only way to change the state is by dispatching an action, which describes what happened.

3. Changes are Made with Pure Reducer Functions: Reducers are pure functions that take the current state and an action as input and return a new state.

Why Use Redux?

Compared to the Context API, Redux offers:

- Predictability: Strict rules ensure state changes are traceable and consistent.

- Debugging Tools: Redux DevTools allow you to inspect state changes, replay actions, and debug efficiently.

- Scalability: Ideal for large apps with complex state interactions.

- Middleware Support: Middleware like `redux-thunk` or `redux-saga` enables handling asynchronous logic (e.g., API calls).

However, Redux has a steeper learning curve and more boilerplate than Context API, so it's best for apps where state complexity justifies the overhead.

Key Concepts in Redux

1. Store: A single object that holds the entire application state.

2. Actions: Plain JavaScript objects with a `type` property (and optional `payload`) that describe what happened (e.g., `{ type: 'INCREMENT', payload: 1 }`).

3. Reducers: Pure functions that specify how the state changes in response to an action. They take the current state and an action, returning a new state.

4. Dispatch: A method to send actions to the store to update the state.

5. Selectors: Functions to extract specific pieces of state from the store.

Redux Workflow

1. A component dispatches an action using the `dispatch` function.

2. The store runs the reducer with the current state and the action.

3. The reducer returns a new state, updating the store.

4. Components subscribed to the store (via hooks or `connect`) re-render with the updated state.

Redux Toolkit

Modern Redux applications use Redux Toolkit, which simplifies setup and reduces boilerplate. It provides utilities like `createSlice` for reducers and `configureStore` for the store, and it includes `redux-thunk` for async operations by default.

Example: Todo List with Redux

Let's build a simple Todo List application using Redux Toolkit to demonstrate state management. This example assumes you're familiar with React components, hooks (`useState`, `useEffect`), and forms/events from your studied topics.

Step 1: Setup

Install the necessary packages:

npm install @reduxjs/toolkit react-redux

Step 2: Create the Redux Slice

A slice in Redux Toolkit combines actions and reducers. Create a file `src/features/todos/todosSlice.js`:

```
import { createSlice } from '@reduxjs/toolkit';

const todosSlice = createSlice({
  name: 'todos', // Slice name
  initialState: {
    items: [], // Array to store todos
  },
  reducers: {
    addTodo: (state, action) => {
      state.items.push({ id: Date.now(), text: action.payload, completed: false });
    },
    toggleTodo: (state, action) => {
      const todo = state.items.find((todo) => todo.id === action.payload);
      if (todo) {
        todo.completed = !todo.completed;
      }
    },
  },
```

```
});

// Export actions

export const { addTodo, toggleTodo } = todosSlice.actions;

// Export reducer

export default todosSlice.reducer;
```

Explanation:

- `createSlice` generates action creators (`addTodo`, `toggleTodo`) and a reducer.

- `initialState` defines the initial state with an empty `items` array.

- Reducers (`addTodo`, `toggleTodo`) mutate the state directly (Redux Toolkit uses Immer to handle immutability under the hood).

- `action.payload` carries data, like the todo text or ID.

Step 3: Configure the Store

Create a store in `src/app/store.js`:

```
import { configureStore } from '@reduxjs/toolkit';

import todosReducer from '../features/todos/todosSlice';

export const store = configureStore({

  reducer: {

    todos: todosReducer, // Mount the todos reducer under the 'todos' key

  },

});
```

Explanation:

- `configureStore` sets up the Redux store with the `todos` reducer.

- The store holds the entire app state, accessible via the `todos` key (e.g., `state.todos.items`).

Step 4: Connect Redux to React

Wrap your app with the `Provider` component in `src/index.js`:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import { Provider } from 'react-redux';
import { store } from './app/store';
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <Provider store={store}>
    <App />
  </Provider>
);
```

Explanation:

- The `Provider` component makes the Redux store available to all components via React's Context API.

- This is similar to how you'd use Context API directly but integrates Redux's store.

Step 5: Create the Todo Component

Create a `TodoList` component in `src/components/TodoList.js` that uses Redux hooks to interact with the store:

```
import React, { useState } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { addTodo, toggleTodo } from '../features/todos/todosSlice';
function TodoList() {
  const [todoText, setTodoText] = useState(''); // Local state for form input
  const todos = useSelector((state) => state.todos.items); // Select todos from store
  const dispatch = useDispatch(); // Get dispatch function
  const handleAddTodo = (e) => {
    e.preventDefault();
    if (todoText.trim()) {
```

```
    dispatch(addTodo(todoText)); // Dispatch action to add todo
    setTodoText(''); // Clear input
  }
 };
 return (
  <div>
   <h2>Todo List</h2>
   <form onSubmit={handleAddTodo}>
    <input
     type="text"
     value={todoText}
     onChange={(e) => setTodoText(e.target.value)}
     placeholder="Add a todo"
    />
    <button type="submit">Add</button>
   </form>
   <ul>
    {todos.map((todo) => (
     <li
      key={todo.id}
      style={{ textDecoration: todo.completed ? 'line-through' : 'none' }}
      onClick={() => dispatch(toggleTodo(todo.id))} // Dispatch action to toggle todo
     >
      {todo.text}
     </li>
    ))}
   </ul>
  </div>
 );
}
```

```
export default TodoList;
```

Explanation:

- useSelector: Extracts the `items` array from the store (`state.todos.items`).
- useDispatch: Provides the `dispatch` function to send actions to the store.
- The form uses local `useState` for the input field, similar to your knowledge of form handling.
- Clicking a todo dispatches the `toggleTodo` action with the todo's ID as the payload.
- The component re-renders automatically when the store updates, thanks to `useSelector`.

Step 6: Use the Component in App

In `src/App.js`:

```
import React from 'react';
import TodoList from './components/TodoList';
function App() {
  return (
    <div>
      <h1>My Redux Todo App</h1>
      <TodoList />
    </div>
  );
}
export default App;
```

How It Works

1. The user types a todo and submits the form, triggering `handleAddTodo`.
2. `dispatch(addTodo(todoText))` sends an action to the store.
3. The `addTodo` reducer adds the new todo to `state.items`.
4. The component re-renders with the updated todos list, accessed via `useSelector`.
5. Clicking a todo dispatches `toggleTodo`, toggling its `completed` status.

Comparison with Context API

- Context API: Simpler, built into React, good for lightweight state sharing (e.g., theme, user data). You've studied this, so you know it uses `createContext` and `useContext`.

- Redux: More structured, better for complex state logic, with tools like Redux DevTools. It avoids prop drilling like Context API but adds boilerplate.

When to Use Redux

- Use Redux when your app has complex state interactions (e.g., multiple components updating shared state).

- For simpler cases, Context API with `useReducer` (which you've studied) might suffice, as it mimics Redux's reducer pattern with less setup.

Additional Tips

- Async Actions: Use `redux-thunk` (included in Redux Toolkit) for API calls. Create async thunks with `createAsyncThunk`.

- Debugging: Install Redux DevTools browser extension to track actions and state changes.

- Performance: Use `useSelector` with specific selectors to avoid unnecessary re-renders, similar to optimizing `useMemo` (from your advanced hooks knowledge).

Key Takeaways

- Redux centralizes state in a store, using actions and reducers to manage updates.

- Redux Toolkit simplifies Redux with utilities like `createSlice` and `configureStore`.

- Compared to Context API, Redux is more powerful for complex apps but requires more setup.

- Hooks like `useSelector` and `useDispatch` integrate Redux with functional components.

This example leverages your knowledge of functional components, hooks (`useState` for the form, `useEffect` could be added for side effects), and form handling, while introducing Redux's structured approach. If you'd like to dive deeper into a specific aspect (e.g., async actions, middleware, or Redux with React Router), let me know!

PERFORMANCE OPTIMIZATION IN REACT

What is Performance Optimization?

Performance optimization in React involves techniques to reduce unnecessary renders, minimize computation, and improve the speed and responsiveness of your application. React is fast by default, but as applications grow in complexity (e.g., with many components, large state objects, or frequent updates), unoptimized code can lead to slow renders, laggy UI, or excessive memory usage. Optimization ensures your app remains performant, especially for large-scale applications using state management (e.g., Context API or Redux).

Why Optimize?

- Improved User Experience: Faster rendering and smoother interactions enhance usability.

- Scalability: Optimized apps handle growing complexity without performance degradation.

- Resource Efficiency: Reduces CPU and memory usage, critical for mobile or low-powered devices.

Key Areas of Focus

1. Preventing Unnecessary Renders: Avoid re-rendering components when their props or state haven't changed.

2. Optimizing Expensive Computations: Cache results of heavy calculations to avoid redundant work.

3. Efficient State Management: Minimize state updates and ensure efficient state access (e.g., in Redux or Context API).

4. Lazy Loading and Code Splitting: Load only the necessary components or code when needed.

5. Event Handling: Optimize event listeners to prevent performance bottlenecks.

Core Optimization Techniques

Below, I'll explain key techniques, leveraging your knowledge of hooks like `useMemo`, `useCallback`, and `useRef`, and connect them to state management (Context API, Redux) and routing.

1. Memoizing Components with `React.memo`

`React.memo` is a higher-order component that prevents a functional component from re-rendering if its props haven't changed. This is especially useful for components that receive the same props frequently.

- When to Use: For components that render often but don't need to update unless their props change.

- Relation to Your Knowledge: Similar to how `useMemo` caches values, `React.memo` caches the component's render output.

2. Memoizing Values with `useMemo`

The `useMemo` hook (which you've studied) caches the result of an expensive computation, recomputing only when dependencies change.

- When to Use: For expensive calculations (e.g., filtering large lists, complex transformations).

- Example Use Case: Filtering a todo list based on user input.

3. Memoizing Functions with `useCallback`

The `useCallback` hook (also studied) caches a function instance, preventing it from being recreated on every render. This is critical when passing callbacks to child components or using them in dependency arrays.

- When to Use: For functions passed as props to `React.memo`-wrapped components or used in `useEffect`/`useMemo` dependencies.

- Relation to Your Knowledge: Prevents unnecessary renders in components that rely on stable function references.

4. Optimizing State Updates

- Local State (`useState`, `useReducer`): Avoid updating state unnecessarily. For example, use `useReducer` (which you've studied) for complex state logic to batch updates.

- Context API: Minimize re-renders by splitting context into smaller, focused providers (e.g., separate contexts for theme and user data).

- Redux: Use specific selectors with `useSelector` to avoid re-renders when unrelated state changes. Redux Toolkit's `createSelector` (from `reselect`) memoizes selectors for efficiency.

5. Lazy Loading and Code Splitting

Using React Router (which you've studied), you can implement code splitting with `React.lazy` and `Suspense` to load components only when needed, reducing initial bundle size.

- When to Use: For large apps with multiple routes or heavy components.

- Example: Load a dashboard component only when the user navigates to the `/dashboard` route.

6. Optimizing Event Handlers

Leverage your knowledge of handling forms and events to optimize event listeners. For example, debounce or throttle expensive event handlers (e.g., input changes for search) to reduce the frequency of updates.

7. Using `useRef` for Persistent Values

The `useRef` hook (studied) can store mutable values that don't trigger re-renders, useful for tracking previous values or DOM references.

- When to Use: To avoid re-renders when storing values like timers or DOM elements.

8. Avoiding Overuse of Optimization

Over-optimizing (e.g., wrapping every component in `React.memo` or overusing `useMemo`) can add complexity and increase memory usage. Optimize only when you identify performance bottlenecks, using tools like React DevTools Profiler.

Example: Optimizing a Todo List Application

Let's build an optimized version of a Todo List application, similar to the Redux example, but focusing on performance optimization techniques. We'll use a combination of `React.memo`, `useMemo`, `useCallback`, `useRef`, and Redux with specific selectors. This example assumes a Redux setup like the one in the previous response.

Step 1: Setup

Assume the same Redux setup from the previous response (`todosSlice.js`, `store.js`, and `Provider` in `index.js`). We'll focus on optimizing the `TodoList` component and its child components.

Step 2: Create a Memoized Todo Item Component

Create a `TodoItem` component in `src/components/TodoItem.js` and wrap it with `React.memo` to prevent unnecessary renders:

```
import React from 'react';
import { useDispatch } from 'react-redux';
import { toggleTodo } from '../features/todos/todosSlice';
const TodoItem = React.memo(({ id, text, completed }) => {
  const dispatch = useDispatch();
  console.log(`Rendering TodoItem: ${text}`); // For debugging renders
  return (
    <li
      style={{ textDecoration: completed ? 'line-through' : 'none' }}
      onClick={() => dispatch(toggleTodo(id))}
    >
      {text}
    </li>
  );
});
export default TodoItem;
```

Explanation:

- `React.memo` ensures `TodoItem` only re-renders if `id`, `text`, or `completed` changes.

- Without `React.memo`, every parent re-render would re-render all `TodoItem` components, even if their props are unchanged.

Step 3: Optimize the Todo List Component

Update `src/components/TodoList.js` to use `useMemo`, `useCallback`, `useRef`, and specific Redux selectors:

```
import React, { useState, useCallback, useMemo, useRef } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { createSelector } from 'reselect';
import { addTodo, toggleTodo } from '../features/todos/todosSlice';
import TodoItem from './TodoItem';
// Memoized selector to avoid unnecessary re-renders
const selectTodos = createSelector(
  [(state) => state.todos.items],
  (items) => items
);
function TodoList() {
  const [todoText, setTodoText] = useState('');
  const dispatch = useDispatch();
  const todos = useSelector(selectTodos); // Use memoized selector
  const inputRef = useRef(null); // Reference to input element
  // Memoize the handleAddTodo function
  const handleAddTodo = useCallback(
    (e) => {
      e.preventDefault();
      if (todoText.trim()) {
        dispatch(addTodo(todoText));
        setTodoText('');
        inputRef.current.focus(); // Focus input after adding
      }
```

```
  },
  [todoText, dispatch]
);
// Memoize filtered todos (e.g., only incomplete todos)
const incompleteTodos = useMemo(() => {
  console.log('Computing incomplete todos'); // For debugging
  return todos.filter((todo) => !todo.completed);
}, [todos]);
return (
  <div>
    <h2>Todo List</h2>
    <form onSubmit={handleAddTodo}>
      <input
        ref={inputRef}
        type="text"
        value={todoText}
        onChange={(e) => setTodoText(e.target.value)}
        placeholder="Add a todo"
      />
      <button type="submit">Add</button>
    </form>
    <h3>All Todos</h3>
    <ul>
      {todos.map((todo) => (
        <TodoItem
          key={todo.id}
          id={todo.id}
          text={todo.text}
          completed={todo.completed}
        />
```

```
      ))}
    </ul>
    <h3>Incomplete Todos</h3>
    <ul>
      {incompleteTodos.map((todo) => (
        <TodoItem
          key={todo.id}
          id={todo.id}
          text={todo.text}
          completed={todo.completed}
        />
      ))}
    </ul>
  </div>
 );
}
export default TodoList;
```

Explanation:

- Memoized Selector (`createSelector`): The `selectTodos` selector memoizes the `todos.items` array, preventing re-renders if other parts of the state change. This builds on your Redux knowledge.

- useCallback: The `handleAddTodo` function is memoized to prevent recreation on every render, ensuring stability when passed to the form's `onSubmit`.

- useMemo: The `incompleteTodos` array is memoized to avoid recomputing the filtered list unless `todos` changes. This leverages your knowledge of `useMemo` for expensive computations.

- useRef: The `inputRef` stores a reference to the input element, allowing focus after adding a todo without triggering a re-render.

- TodoItem with `React.memo`: Each `TodoItem` only re-renders if its props change, optimizing the list rendering.

Step 4: Add Code Splitting with React Router

To demonstrate lazy loading (building on your React Router knowledge), let's assume the `TodoList` component is loaded lazily in `src/App.js`:

```
import React, { Suspense } from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import { Provider } from 'react-redux';
import { store } from './app/store';
// Lazy load TodoList
const TodoList = React.lazy(() => import('./components/TodoList'));
function App() {
  return (
    <Provider store={store}>
      <Router>
        <div>
          <h1>My Optimized Todo App</h1>
          <Suspense fallback={<div>Loading...</div>}>
            <Routes>
              <Route path="/" element={<TodoList />} />
            </Routes>
          </Suspense>
        </div>
      </Router>
    </Provider>
  );
}
export default App;
```

Explanation:

- React.lazy: Loads `TodoList` only when the `/` route is accessed, reducing the initial bundle size.

- Suspense: Displays a fallback UI (`Loading...`) while the component loads.

- This leverages your knowledge of React Router for dynamic imports.

Step 5: Optimizing Event Handlers

To optimize form input handling (building on your forms/events knowledge), add debouncing to the input's `onChange` event using a library like `lodash.debounce`. First, install Lodash:

npm install lodash

Update the input in `TodoList.js`:

```
import React, { useState, useCallback, useMemo, useRef } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { createSelector } from 'reselect';
import { addTodo, toggleTodo } from '../features/todos/todosSlice';
import { debounce } from 'lodash';
import TodoItem from './TodoItem';
const selectTodos = createSelector(
  [(state) => state.todos.items],
  (items) => items
);
function TodoList() {
  const [todoText, setTodoText] = useState('');
  const dispatch = useDispatch();
  const todos = useSelector(selectTodos);
  const inputRef = useRef(null);
  // Debounced input handler
  const handleInputChange = useCallback(
    debounce((value) => {
      setTodoText(value);
    }, 300),
    []
  );
  const handleAddTodo = useCallback(
    (e) => {
      e.preventDefault();
      if (todoText.trim()) {
```

```
      dispatch(addTodo(todoText));
      setTodoText('');
      inputRef.current.focus();
    }
  },
  [todoText, dispatch]
);
const incompleteTodos = useMemo(() => {
  console.log('Computing incomplete todos');
  return todos.filter((todo) => !todo.completed);
}, [todos]);
return (
  <div>
    <h2>Todo List</h2>
    <form onSubmit={handleAddTodo}>
      <input
        ref={inputRef}
        type="text"
        value={todoText}
        onChange={(e) => handleInputChange(e.target.value)}
        placeholder="Add a todo"
      />
      <button type="submit">Add</button>
    </form>
    <h3>All Todos</h3>
    <ul>
      {todos.map((todo) => (
        <TodoItem
          key={todo.id}
          id={todo.id}
```

```
          text={todo.text}
          completed={todo.completed}
        />
      ))}
    </ul>
    <h3>Incomplete Todos</h3>
    <ul>
      {incompleteTodos.map((todo) => (
        <TodoItem
          key={todo.id}
          id={todo.id}
          text={todo.text}
          completed={todo.completed}
        />
      ))}
    </ul>
  </div>
 );
}
export default TodoList;
```

Explanation:

- Debouncing: The `handleInputChange` function is debounced to limit how often `setTodoText` is called during rapid typing, reducing state updates and re-renders.

- This builds on your knowledge of form handling, optimizing for performance.

How It Works

1. Component Memoization: `TodoItem` components only re-render when their props (`id`, `text`, `completed`) change, thanks to `React.memo`.

2. Selector Memoization: The `selectTodos` selector prevents re-renders when unrelated state changes, optimizing Redux usage.

3. Function Memoization: `handleAddTodo` and `handleInputChange` are memoized with `useCallback`, ensuring stable references for event handlers and dependencies.

4. Value Memoization: `incompleteTodos` is computed only when `todos` changes, thanks to `useMemo`.

5. Ref Usage: `inputRef` focuses the input without triggering re-renders.

6. Lazy Loading: The `TodoList` component is loaded only when needed, reducing initial load time.

7. Debounced Input: The input's `onChange` event is debounced, reducing state updates during typing.

Performance Benefits

- Reduced Renders: `React.memo` and `createSelector` prevent unnecessary re-renders of `TodoItem` and `TodoList`.

- Efficient Computations: `useMemo` avoids recalculating `incompleteTodos` unnecessarily.

- Stable Callbacks: `useCallback` ensures event handlers don't trigger child re-renders.

- Smaller Bundle: Lazy loading with `React.lazy` reduces initial JavaScript load.

- Optimized Events: Debouncing reduces state updates for rapid input changes.

Key Takeaways

- Memoization: Use `React.memo`, `useMemo`, and `useCallback` to cache components, values, and functions, preventing unnecessary work.

- State Management: Optimize Redux with `createSelector` and specific `useSelector` calls; for Context API, split contexts to reduce re-renders.

- Lazy Loading: Use `React.lazy` and `Suspense` with React Router for code splitting.

- Event Optimization: Debounce or throttle event handlers for performance-critical interactions.

- Measure First: Use React DevTools Profiler to identify bottlenecks before optimizing.

This example ties together your knowledge of hooks (`useMemo`, `useCallback`, `useRef`), Redux, forms, and React Router, showing how to apply performance optimizations in a practical scenario. If you want to dive deeper into a specific technique (e.g., advanced Redux optimizations, code splitting with multiple routes, or profiling with React DevTools), let me know!

TESTING IN REACT

What is Testing in React?

Testing in React involves verifying that your components, hooks, state management, and application logic behave as expected. It ensures your app is reliable, maintainable, and resistant to regressions when code changes. Testing is critical for large-scale applications, especially when using complex state management (e.g., Redux, Context API) or routing (React Router), as it validates interactions across components and features.

Why Test?

- Reliability: Ensures components render correctly and handle user interactions as intended.

- Refactoring Safety: Allows confident code changes without breaking functionality.

- Scalability: Maintains code quality in large apps with many components and state interactions.

- User Experience: Prevents bugs that could degrade the user experience.

Types of Tests in React

1. Unit Tests: Test individual components, hooks, or functions in isolation (e.g., a single component or reducer).

2. Integration Tests: Test interactions between components, such as a form dispatching Redux actions or navigating with React Router.

3. End-to-End (E2E) Tests: Test the entire application flow, simulating real user interactions (e.g., filling a form and navigating routes).

4. Snapshot Tests: Capture a component's rendered output and compare it to a stored snapshot to detect unintended changes.

Popular Testing Tools

- Jest: A JavaScript testing framework (included with Create React App) for running tests, assertions, and mocking.

- React Testing Library: A library for testing React components by simulating user interactions, focusing on testing behavior rather than implementation details.

- Enzyme (less common now): An older library for testing React components, often used for shallow rendering.

- Cypress or Playwright: For E2E testing, simulating real user flows in a browser.

- Redux Testing Utilities: Tools like `@redux-devtools` or Jest mocks for testing Redux actions and reducers.

- React Router Testing: Utilities like `MemoryRouter` from `react-router-dom` for testing routing logic.

Key Principles of React Testing

- Test Behavior, Not Implementation: Focus on what the component does (e.g., renders a button, updates state) rather than how it's implemented (e.g., internal state logic). React Testing Library encourages this approach.

- Simulate User Interactions: Use tools like `fireEvent` or `userEvent` to mimic clicks, typing, etc., aligning with your knowledge of form/event handling.

- Mock Dependencies: Mock external dependencies (e.g., API calls, Redux store, React Router) to isolate tests.

- Cover Edge Cases: Test loading states, error conditions, and empty states, especially for components using Redux or Context API.

Testing with Your Studied Topics

- Components and Props: Test that components render correctly with different prop values.

- State and Hooks: Verify that `useState`, `useEffect`, `useContext`, etc., update the UI as expected.

- Forms and Events: Test form submissions, input changes, and event handlers.

- Redux: Test reducers, actions, and components using `useSelector`/`useDispatch`.

- React Router: Test navigation and route-specific rendering.

- Performance Optimization: Ensure optimizations (e.g., `useMemo`, `React.memo`) don't break functionality.

Example: Testing a Todo List Component

Let's create a testable version of a Todo List component (similar to the Redux and performance optimization examples) and write unit and integration tests using Jest and React Testing Library. The component will use Redux for state management, React Router for navigation, and hooks for local state and side effects, leveraging your studied topics.

Step 1: Setup

Install the necessary testing libraries (if using Create React App, Jest and React Testing Library are included):

npm install --save-dev @testing-library/react @testing-library/jest-dom @testing-library/user-event

Add `jest-dom` to your test setup file (e.g., `src/setupTests.js`):

```
import '@testing-library/jest-dom';
```

Step 2: Create the Component

Here's a `TodoList` component in `src/components/TodoList.js` that uses Redux, hooks, and React Router:

```
import React, { useState, useEffect } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { Link } from 'react-router-dom';
import { addTodo, toggleTodo } from '../features/todos/todosSlice';
// Assume todosSlice.js is the same as in the Redux example
function TodoList() {
 const [todoText, setTodoText] = useState('');
 const todos = useSelector((state) => state.todos.items);
 const dispatch = useDispatch();
 useEffect(() => {
  document.title = `Todos: ${todos.length}`; // Update title with todo count
 }, [todos.length]);
 const handleAddTodo = (e) => {
  e.preventDefault();
  if (todoText.trim()) {
   dispatch(addTodo(todoText));
   setTodoText('');
  }
 };
 return (
  <div data-testid="todo-list">
   <h2>Todo List</h2>
   <form onSubmit={handleAddTodo}>
    <input
     data-testid="todo-input"
     type="text"
     value={todoText}
     onChange={(e) => setTodoText(e.target.value)}
```

```
          placeholder="Add a todo"
        />
        <button type="submit" data-testid="add-button">
          Add
        </button>
      </form>
      <ul data-testid="todo-items">
        {todos.map((todo) => (
          <li
            key={todo.id}
            data-testid={`todo-item-${todo.id}`}
            style={{ textDecoration: todo.completed ? 'line-through' : 'none' }}
            onClick={() => dispatch(toggleTodo(todo.id))}
          >
            {todo.text}
          </li>
        ))}
      </ul>
      <Link to="/completed" data-testid="completed-link">
        View Completed Todos
      </Link>
    </div>
  );
}
export default TodoList;
```

Explanation:

- Uses Redux (`useSelector`, `useDispatch`) for state management, building on your Redux knowledge.

- Uses hooks (`useState` for form input, `useEffect` for side effects), aligning with your hooks knowledge.

- Includes React Router (`Link`) for navigation, tying into your routing knowledge.

- Adds `data-testid` attributes for easy DOM querying in tests, a common practice with React Testing Library.

Step 3: Write Unit Tests

Create a test file `src/components/TodoList.test.js` to test the component's rendering, form submission, and Redux interactions:

```
import React from 'react';
import { render, screen, fireEvent } from '@testing-library/react';
import { Provider } from 'react-redux';
import { configureStore } from '@reduxjs/toolkit';
import { MemoryRouter } from 'react-router-dom';
import TodoList from './TodoList';
import todosReducer from '../features/todos/todosSlice';
// Create a mock store for testing
const createMockStore = (initialState = { todos: { items: [] } }) => {
  return configureStore({
    reducer: {
      todos: todosReducer,
    },
    preloadedState: initialState,
  });
};
describe('TodoList Component', () => {
  test('renders TodoList component', () => {
    const store = createMockStore();
    render(
      <Provider store={store}>
        <MemoryRouter>
          <TodoList />
        </MemoryRouter>
      </Provider>
```

```
  );
  expect(screen.getByTestId('todo-list')).toBeInTheDocument();
  expect(screen.getByText('Todo List')).toBeInTheDocument();
  expect(screen.getByTestId('todo-input')).toBeInTheDocument();
  expect(screen.getByTestId('add-button')).toBeInTheDocument();
  expect(screen.getByTestId('completed-link')).toBeInTheDocument();
});
test('adds a new todo when form is submitted', () => {
  const store = createMockStore();
  render(
    <Provider store={store}>
      <MemoryRouter>
        <TodoList />
      </MemoryRouter>
    </Provider>
  );
  const input = screen.getByTestId('todo-input');
  const button = screen.getByTestId('add-button');
  fireEvent.change(input, { target: { value: 'Test Todo' } });
  fireEvent.click(button);
  expect(screen.getByText('Test Todo')).toBeInTheDocument();
});
test('toggles a todo when clicked', () => {
  const initialState = {
    todos: {
      items: [{ id: 1, text: 'Test Todo', completed: false }],
    },
  };
  const store = createMockStore(initialState);
  render(
```

```
      <Provider store={store}>
        <MemoryRouter>
          <TodoList />
        </MemoryRouter>
      </Provider>
    );
    const todoItem = screen.getByTestId('todo-item-1');
    expect(todoItem).not.toHaveStyle('text-decoration: line-through');
    fireEvent.click(todoItem);
    expect(todoItem).toHaveStyle('text-decoration: line-through');
  });
  test('renders link to completed todos', () => {
    const store = createMockStore();
    render(
      <Provider store={store}>
        <MemoryRouter>
          <TodoList />
        </MemoryRouter>
      </Provider>
    );
    const link = screen.getByTestId('completed-link');
    expect(link).toHaveAttribute('href', '/completed');
  });
});
```

Explanation:

- Setup: The `createMockStore` function creates a Redux store with the `todosReducer` for testing, using Redux Toolkit's `configureStore`.

- MemoryRouter: Wraps the component to mock React Router's context, allowing testing of the `Link` component.

- Rendering Test: Verifies that the component renders correctly, checking for key elements using `data-testid`.

- Form Submission Test: Simulates typing in the input and clicking the button, checking that the new todo appears. This tests form handling and Redux `addTodo` action.

- Todo Toggle Test: Tests clicking a todo item to toggle its `completed` status, verifying the style change (Redux `toggleTodo` action).

- Link Test: Ensures the `Link` component has the correct `href`, tying into your React Router knowledge.

- React Testing Library: Uses `screen.getByTestId` and `fireEvent` to query the DOM and simulate user interactions, aligning with your forms/events knowledge.

Step 4: Write a Test for useEffect

Add a test to verify the `useEffect` hook that updates the document title:

```
test('updates document title with todo count', () => {
  const initialState = {
    todos: {
      items: [
        { id: 1, text: 'Todo 1', completed: false },
        { id: 2, text: 'Todo 2', completed: false },
      ],
    },
  };
  const store = createMockStore(initialState);
  render(
    <Provider store={store}>
      <MemoryRouter>
        <TodoList />
      </MemoryRouter>
    </Provider>
  );
  expect(document.title).toBe('Todos: 2');
})
```

Explanation:

- Tests the `useEffect` hook's side effect, verifying that `document.title` updates based on the number of todos.

- Uses the mock Redux store to provide an initial state with two todos.

- Builds on your `useEffect` knowledge to test side effects.

Step 5: Testing Redux Reducers

Test the `todosSlice` reducer in `src/features/todos/todosSlice.test.js`:

```
import todosReducer, { addTodo, toggleTodo } from './todosSlice';
describe('todos reducer', () => {
  test('should handle addTodo', () => {
    const initialState = { items: [] };
    const action = addTodo('Test Todo');
    const newState = todosReducer(initialState, action);
    expect(newState.items).toHaveLength(1);
    expect(newState.items[0].text).toBe('Test Todo');
    expect(newState.items[0].completed).toBe(false);
  });
  test('should handle toggleTodo', () => {
    const initialState = {
      items: [{ id: 1, text: 'Test Todo', completed: false }],
    };
    const action = toggleTodo(1);
    const newState = todosReducer(initialState, action);
    expect(newState.items[0].completed).toBe(true);
  });
});
```

Explanation:

- Tests the `addTodo` and `toggleTodo` reducers to ensure they update the state correctly.

- Verifies the shape of the state after dispatching actions, building on your Redux knowledge.

- Tests pure reducer functions in isolation, a core unit testing practice.

Step 6: Mocking API Calls (Optional)

If the `TodoList` component fetched todos from an API using `useEffect`, you could mock the API call. Here's an example assuming an async `fetchTodos` action in `todosSlice.js`:

```
// In todosSlice.js (add this to the slice)
import { createAsyncThunk } from '@reduxjs/toolkit';
export const fetchTodos = createAsyncThunk('todos/fetchTodos', async () => {
  const response = await fetch('/api/todos');
  return response.json();
});
// In TodoList.test.js
import { waitFor } from '@testing-library/react';
jest.mock('axios'); // Mock axios or fetch
import axios from 'axios';
test('fetches todos on mount', async () => {
  const mockTodos = [{ id: 1, text: 'Test Todo', completed: false }];
  axios.get.mockResolvedValue({ data: mockTodos });
  const store = createMockStore();
  render(
    <Provider store={store}>
      <MemoryRouter>
        <TodoList />
      </MemoryRouter>
    </Provider>
  );
  await waitFor(() => {
    expect(screen.getByText('Test Todo')).toBeInTheDocument();
  });
});
```

Explanation:

- Mocks an API call using Jest's mocking capabilities.

- Uses `waitFor` to wait for async operations, testing the `useEffect` that fetches todos.

- Builds on your `useEffect` and Redux knowledge for async actions.

How It Works

1. Rendering: The component renders with a form, todo list, and navigation link, tested for correct DOM structure.

2. Form Interaction: Simulates user input and submission, verifying Redux state updates and UI changes.

3. Redux Integration: Tests ensure Redux actions (`addTodo`, `toggleTodo`) update the store and UI correctly.

4. Routing: The `Link` component is tested for correct navigation behavior.

5. Side Effects: The `useEffect` hook's document title update is verified.

6. Reducer Tests: Pure reducer functions are tested in isolation for predictable state changes.

Best Practices

- Use React Testing Library: Prefer it over Enzyme for its focus on user behavior.

- Mock Sparingly: Only mock external dependencies (e.g., APIs, Redux store) to isolate tests.

- Test Edge Cases: Test empty states, error conditions, and loading states (e.g., no todos, failed API calls).

- Keep Tests Focused: Each test should verify one behavior (e.g., rendering, form submission).

- Use Descriptive Test Names: Clear `describe` and `test` names improve readability.

- Leverage Your Knowledge: Use your understanding of hooks, Redux, and routing to test relevant interactions.

Comparison with Other Topics

- Hooks: Tests verify `useState` (form input), `useEffect` (document title), and Redux hooks (`useSelector`, `useDispatch`).

- Redux: Tests cover reducers and component integration with the store, similar to your Redux knowledge.

- Routing: The `MemoryRouter` enables testing of `Link`, aligning with your React Router knowledge.

- Forms/Events: Simulating input changes and clicks uses your form/event handling skills.

- Performance Optimization: Ensure optimizations (e.g., `useMemo`, `React.memo`) don't break tests by focusing on behavior.

Key Takeaways

- Testing Tools: Use Jest and React Testing Library for unit and integration tests, Cypress/Playwright for E2E tests.

- Test Types: Unit tests for components/reducers, integration tests for component interactions, E2E tests for user flows.

- Behavior Focus: Test what the user sees and does, not internal implementation details.

- Mocking: Mock Redux, APIs, and routing to isolate tests.

- Comprehensive Coverage: Test rendering, state updates, events, side effects, and navigation.

This example leverages your knowledge of components, hooks, Redux, and routing to create and test a realistic React component. If you want to dive deeper into a specific aspect (e.g., E2E testing with Cypress, mocking APIs, or testing optimized components with `useMemo`/`React.memo`), let me know!

ADVANCED TOPICS, PROJECTS, AND DEPLOYMENT IN REACT

What are Advanced Topics, Projects, and Deployment?

- Advanced Topics: These include sophisticated React patterns and tools that enhance scalability, maintainability, and performance, such as custom hooks, render props, higher-order components (HOCs), TypeScript integration, and server-side rendering (SSR) or static site generation (SSG).

- Projects: Structuring a React project involves organizing code, managing dependencies, and integrating features like state management, routing, and testing for scalability and collaboration.

- Deployment: Deploying a React application involves building, optimizing, and hosting the app on a server or platform, ensuring it's accessible, performant, and secure in production.

Why Focus on These?

- Scalability: Advanced patterns and project structure support large, maintainable applications.

- Performance: Techniques like SSR and code splitting enhance user experience.

- Collaboration: Well-organized projects improve team workflows and code quality.

- Production Readiness: Deployment ensures your app is accessible, reliable, and optimized for real users.

Advanced Topics

Here are key advanced React topics, building on your studied concepts:

1. Custom Hooks

- **What:** Reusable functions that encapsulate hook-based logic (e.g., `useState`, `useEffect`) for sharing across components.

- **Why:** Reduces code duplication and improves maintainability, leveraging your hooks knowledge.

- **Example:** A `useForm` hook to manage form state and validation (builds on your forms/events knowledge).

2. Render Props and HOCs

- **Render Props:** A pattern where a component's prop is a function that returns JSX, enabling shared logic.

- **HOCs:** Functions that wrap components to add functionality (e.g., authentication).

- **Why:** Alternatives to hooks for sharing logic, though less common in modern React.

- **Relation:** Complements your knowledge of components and props.

3. TypeScript Integration

- **What:** Adding static types to React components, props, and state for better type safety and developer experience.

- **Why:** Reduces runtime errors and improves IDE support, especially for large projects.

- **Relation:** Enhances your props and state management knowledge.

4. Server-Side Rendering (SSR) and Static Site Generation (SSG)

- **SSR:** Renders React components on the server for faster initial page loads and SEO (e.g., with Next.js).

- **SSG:** Generates static HTML at build time for performance and scalability.

- **Why:** Improves performance and SEO, building on your performance optimization knowledge.

- **Relation:** Uses React Router concepts for dynamic routing in SSR/SSG frameworks.

5. Suspense and Concurrent Rendering

- **Suspense:** Handles asynchronous rendering (e.g., lazy loading, data fetching) with a fallback UI.

- **Concurrent Rendering:** React 18 feature for non-blocking renders, improving responsiveness.

- **Why:** Enhances user experience for dynamic apps, tying into your `React.lazy` and `Suspense` knowledge from performance optimization.

6. Error Boundaries

- What: Components that catch JavaScript errors in their child component tree, preventing app crashes.

- Why: Improves robustness, especially for complex apps with Redux or Context API.

- Relation: Builds on your component knowledge.

Project Structure

A well-organized React project is crucial for scalability and team collaboration. Here's a recommended structure for a large-scale React app, incorporating your knowledge of Redux, React Router, and testing:

```
my-react-app/
├── src/
│   ├── assets/                # Images, fonts, etc.
│   ├── components/            # Reusable components
│   │   ├── common/            # Shared UI (e.g., Button, Input)
│   │   ├── features/          # Feature-specific components (e.g., TodoList)
│   ├── features/              # Redux slices or Context logic
│   │   ├── todos/             # Redux slice (todosSlice.js)
│   ├── pages/                 # Route-specific components (React Router)
│   │   ├── Home.js
│   │   ├── CompletedTodos.js
│   ├── hooks/                 # Custom hooks (e.g., useForm.js)
│   ├── contexts/              # Context API providers
│   ├── utils/                 # Utility functions (e.g., API calls, helpers)
│   ├── routes/                # Route definitions (React Router)
│   ├── App.js                 # Main app with routes
│   ├── index.js               # Entry point with Provider
│   ├── setupTests.js          # Testing setup
│   ├── __tests__/             # Test files
├── public/                    # Static assets (index.html, favicon)
├── package.json               # Dependencies and scripts
├── .env                       # Environment variables
├── README.md                  # Project documentation
```

Key Practices:

- Modularize: Group related code (e.g., Redux slices with feature components).

- Feature-Based: Organize by feature (e.g., `todos/` for slice, components, and tests), aligning with Redux and Context API.

- TypeScript (Optional): Add `.ts` or `.tsx` files for type safety.

- Testing: Colocate tests in `__tests__` or alongside components.

- Environment Variables: Use `.env` for API keys, base URLs, etc.

Deployment

Deploying a React app involves building the app, optimizing the production bundle, and hosting it on a platform. Common steps include:

1. Building for Production

- Run `npm run build` (Create React App) to create an optimized bundle in the `build/` folder.

- Optimizations include minification, tree-shaking, and code splitting (from your performance optimization knowledge).

2. Hosting Platforms

- Vercel: Ideal for React apps, supports Next.js for SSR/SSG, and offers easy deployment.

- Netlify: Great for static React apps, with automatic scaling and CDN.

- AWS Amplify/S3: For serverless hosting or static assets.

- Firebase: Google's platform for hosting and backend integration.

- Heroku: Suitable for server-based apps (e.g., with SSR).

3. Deployment Steps

- Build the app: `npm run build`.

- Deploy to a platform (e.g., `vercel deploy` or `netlify deploy`).

- Configure environment variables (e.g., API URLs).

- Set up a domain or CDN for better performance.

4. Continuous Integration/Continuous Deployment (CI/CD)

- Use GitHub Actions, CircleCI, or Vercel/Netlify's built-in CI/CD to automate testing and deployment.

- Run tests (from your testing knowledge) before deployment to ensure quality.

5. Performance and SEO

- Use SSR/SSG for SEO (e.g., Next.js).

- Optimize images and assets (from your performance optimization knowledge).

- Enable caching and CDN for faster load times.

Challenges and Best Practices

- Advanced Topics: Avoid overusing complex patterns (e.g., HOCs) when hooks suffice, as hooks are more idiomatic in modern React.

- Project Structure: Keep consistency across teams, document conventions, and use linters (e.g., ESLint) for code quality.

- Deployment: Monitor performance post-deployment using tools like Lighthouse or Sentry for error tracking.

- Testing Integration: Ensure tests cover advanced features (e.g., custom hooks, SSR), building on your testing knowledge.

Example: Advanced Todo App with Deployment

Let's build an advanced Todo app that incorporates custom hooks, TypeScript, error boundaries, Redux, React Router, and deployment. The example will be simplified but realistic, demonstrating advanced topics and project structure, with deployment to Vercel.

Step 1: Project Structure

Set up the project with TypeScript and a modular structure:

```
todo-app/
├── src/
│   ├── assets/                # Images, icons
│   ├── components/
│   │   ├── common/
│   │   │   ├── ErrorBoundary.tsx
│   │   ├── features/
│   │   │   ├── TodoList.tsx
│   │   │   ├── TodoItem.tsx
│   ├── features/
│   │   ├── todos/
│   │   │   ├── todosSlice.ts
│   ├── hooks/
│   │   ├── useForm.ts
│   ├── pages/
│   │   ├── Home.tsx
│   │   ├── CompletedTodos.tsx
│   ├── routes/
│   │   ├── index.tsx
│   ├── App.tsx
│   ├── index.tsx
│   ├── __tests__/
│   │   ├── TodoList.test.tsx
├── public/
├── package.json
├── tsconfig.json
├── .env
├── README.md
```

Install dependencies:

npm install @reduxjs/toolkit react-redux react-router-dom typescript @types/react @types/react-dom axios

Step 2: Custom Hook (useForm)

Create a custom hook in `src/hooks/useForm.ts` for form management:

```
import { useState, ChangeEvent, FormEvent } from 'react';
interface FormState<T> {
  values: T;
  handleChange: (e: ChangeEvent<HTMLInputElement>) => void;
  handleSubmit: (callback: (values: T) => void) => (e: FormEvent) => void;
  reset: () => void;
}
function useForm<T>(initialValues: T): FormState<T> {
  const [values, setValues] = useState<T>(initialValues);
  const handleChange = (e: ChangeEvent<HTMLInputElement>) => {
    setValues((prev) => ({ ...prev, [e.target.name]: e.target.value }));
  };
  const handleSubmit = (callback: (values: T) => void) => (e: FormEvent) => {
    e.preventDefault();
    callback(values);
  };
  const reset = () => setValues(initialValues);
  return { values, handleChange, handleSubmit, reset };
}
export default useForm;
```

Explanation:

- A generic TypeScript hook that manages form state and events, building on your `useState` and forms knowledge.

- Reusable across components, reducing boilerplate.

Step 3: Error Boundary

Create an error boundary in `src/components/common/ErrorBoundary.tsx`:

```
import React, { Component, ReactNode } from 'react';
interface Props {
  children: ReactNode;
}
interface State {
  hasError: boolean;
}
class ErrorBoundary extends Component<Props, State> {
  state: State = { hasError: false };
  static getDerivedStateFromError(): State {
    return { hasError: true };
  }
  componentDidCatch(error: Error, errorInfo: React.ErrorInfo) {
    console.error('Error:', error, errorInfo);
  }
  render() {
    if (this.state.hasError) {
      return <h2>Something went wrong. Please try again.</h2>;
    }
    return this.props.children;
  }
}
export default ErrorBoundary;
```

Explanation:

- Catches errors in child components, preventing app crashes.

- Uses class components (from your functional vs. class knowledge) since error boundaries require lifecycle methods.

Step 4: TodoList Component with TypeScript and Redux

Update `src/components/features/TodoList.tsx` to use TypeScript, Redux, and the custom `useForm` hook:

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { Link } from 'react-router-dom';
import { addTodo, toggleTodo } from '../../features/todos/todosSlice';
import useForm from '../../hooks/useForm';
import ErrorBoundary from '../common/ErrorBoundary';
interface Todo {
  id: number;
  text: string;
  completed: boolean;
}
interface RootState {
  todos: { items: Todo[] };
}
function TodoList() {
  const { values, handleChange, handleSubmit } = useForm({ todoText: '' });
  const todos = useSelector((state: RootState) => state.todos.items);
  const dispatch = useDispatch();
  const onSubmit = handleSubmit(({ todoText }) => {
    if (todoText.trim()) {
      dispatch(addTodo(todoText));
    }
  });
  return (
    <ErrorBoundary>
      <div data-testid="todo-list">
        <h2>Todo List</h2>
```

```
      <form onSubmit={onSubmit}>
        <input
          type="text"
          name="todoText"
          value={values.todoText}
          onChange={handleChange}
          placeholder="Add a todo"
          data-testid="todo-input"
        />
        <button type="submit" data-testid="add-button">
          Add
        </button>
      </form>
      <ul data-testid="todo-items">
        {todos.map((todo) => (
          <li
            key={todo.id}
            data-testid={`todo-item-${todo.id}`}
            style={{ textDecoration: todo.completed ? 'line-through' : 'none' }}
            onClick={() => dispatch(toggleTodo(todo.id))}
          >
            {todo.text}
          </li>
        ))}
      </ul>
      <Link to="/completed" data-testid="completed-link">
        View Completed Todos
      </Link>
    </div>
  </ErrorBoundary>
```

```
  );
}
export default TodoList;
```

Explanation:

- Uses TypeScript for type safety (e.g., `Todo`, `RootState` interfaces).
- Integrates Redux (`useSelector`, `useDispatch`) for state management.
- Uses the custom `useForm` hook for form handling, reducing boilerplate.
- Wraps in ErrorBoundary for robustness.
- Includes React Router (`Link`) for navigation.

Step 5: Routing with React Router

Set up routes in `src/routes/index.tsx`:

```
import React, { lazy, Suspense } from 'react';
import { Routes, Route } from 'react-router-dom';
const TodoList = lazy(() => import('../components/features/TodoList'));
const CompletedTodos = lazy(() => import('../pages/CompletedTodos'));
function AppRoutes() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <Routes>
        <Route path="/" element={<TodoList />} />
        <Route path="/completed" element={<CompletedTodos />} />
      </Routes>
    </Suspense>
  );
}
export default AppRoutes;
```

Explanation:

- Uses lazy loading with `Suspense` (from your performance optimization knowledge) to reduce bundle size.

- Defines routes for the home page and completed todos, building on your React Router knowledge.

Step 6: App Entry Point

In `src/App.tsx`:

```
import React from 'react';
import { Provider } from 'react-redux';
import { BrowserRouter } from 'react-router-dom';
import { store } from './app/store';
import AppRoutes from './routes';
function App() {
  return (
    <Provider store={store}>
      <BrowserRouter>
        <div>
          <h1>My Advanced Todo App</h1>
          <AppRoutes />
        </div>
      </BrowserRouter>
    </Provider>
  );
}
export default App;
```

Step 7: Deployment to Vercel

1. Build the App:

- Ensure a `vercel.json` file (optional) for configuration:

```json
{
  "rewrites": [{ "source": "/(.*)", "destination": "/index.html" }]
}
```

- Run `npm run build` to create the production bundle.

2. Deploy:

- Install Vercel CLI: `npm install -g vercel`.
- Run `vercel` in the project root and follow prompts to deploy.
- Configure environment variables (e.g., API keys) in Vercel's dashboard.

3. CI/CD:

- Connect the project to a GitHub repository.
- Vercel automatically builds and deploys on `git push`.

4. Optimize:

- Use Vercel's CDN and caching for performance.
- Monitor with Vercel Analytics or integrate Sentry for error tracking.

Step 8: Testing (Optional)

Add a test in `src/__tests__/TodoList.test.tsx` to verify the component, leveraging your testing knowledge:

```
import React from 'react';
import { render, screen, fireEvent } from '@testing-library/react';
import { Provider } from 'react-redux';
import { configureStore } from '@reduxjs/toolkit';
import { MemoryRouter } from 'react-router-dom';
import TodoList from '../components/features/TodoList';
import todosReducer from '../features/todos/todosSlice';
const createMockStore = () =>
  configureStore({
    reducer: { todos: todosReducer },
```

```
    preloadedState: { todos: { items: [] } },
  });
describe('TodoList', () => {
  test('adds a todo', () => {
    const store = createMockStore();
    render(
      <Provider store={store}>
        <MemoryRouter>
          <TodoList />
        </MemoryRouter>
      </Provider>
    );
    fireEvent.change(screen.getByTestId('todo-input'), { target: { value: 'Test Todo' } });
    fireEvent.click(screen.getByTestId('add-button'));
    expect(screen.getByText('Test Todo')).toBeInTheDocument();
  });
});
```

Explanation:

- Tests the `TodoList` component's form submission, ensuring Redux integration works.

- Uses `MemoryRouter` and a mock Redux store, aligning with your testing and routing knowledge.

How It Works

1. Advanced Topics:

   - Custom Hook: `useForm` simplifies form logic across components.

   - TypeScript: Ensures type safety for props, state, and Redux.

   - Error Boundary: Protects the app from crashes.

   - Suspense: Enables lazy loading for routes.

2. Project Structure: Modular organization by features, hooks, and pages.

3. Deployment: Vercel hosts the optimized build, with CI/CD for automation.

4. Testing: Verifies functionality, ensuring reliability in production.

Key Features Demonstrated

- Custom Hooks: `useForm` for reusable form logic.

- TypeScript: Type-safe components and Redux state.

- Error Boundaries: Robust error handling.

- Lazy Loading: Optimized routing with `React.lazy` and `Suspense`.

- Redux: Centralized state management.

- React Router: Dynamic navigation.

- Deployment: Production-ready hosting on Vercel.

Key Takeaways

- Advanced Topics: Use custom hooks, TypeScript, error boundaries, and SSR/SSG for scalable, robust apps.

- Project Structure: Organize by features, colocate related code, and use TypeScript for type safety.

- Deployment: Build and deploy with platforms like Vercel, leveraging CI/CD and performance optimizations.

- Integration with Studied Topics:

  - Hooks: Custom hooks extend `useState`, `useEffect`, etc.

  - Redux/Context: Advanced state management integrates with TypeScript.

  - Routing: Lazy loading enhances React Router performance.

  - Testing: Ensures advanced features work as expected.

  - Performance: Code splitting and SSR build on optimization techniques.

This example ties together your knowledge into a production-ready app. If you want to dive deeper into a specific area (e.g., Next.js for SSR, advanced TypeScript patterns, or CI/CD pipelines), let me know!