Complete Beginner's Guide to React with TypeScript

Table of Contents

1. Introduction to React
2. Setting Up with Vite + TypeScript
3. JSX and TSX Basics
4. Creating Components
5. Props and State
6. Event Handling
7. Conditional Rendering
8. Lists and Keys
9. Hooks and Lifecycle with useEffect
10. Fetching Data
11. Handling Form Inputs and Two-Way Binding
12. Forms with React Hook Form
13. Routing with React Router v6+
14. Mini Project: Task Manager App

---

1. Introduction to React

What is React?

React is a JavaScript library for building user interfaces, especially web applications. Think of it as a tool that helps you create interactive websites more efficiently. Instead of manually updating parts of your webpage when data changes, React does this automatically for you.

Why React?

Traditional Web Development:

```
User clicks button → Manually find HTML element → Update content → Handle
edge cases
```

React Way:

```
User clicks button → Update data → React automatically updates the UI
```

Key Concepts

Components: Think of components as custom HTML elements. Instead of writing repetitive HTML, you create reusable pieces.

Virtual DOM: React creates a lightweight copy of your webpage in memory. When something changes, React figures out the most efficient way to update the real webpage.

Declarative: You describe what you want the UI to look like, not how to achieve it step by step.

The React Ecosystem

- **React**: The core library
- **Vite**: Fast build tool (replaces Create React App)
- **TypeScript**: Adds type safety to JavaScript
- **React Hook Form**: Makes form handling easier
- **React Router**: Handles navigation between pages

🎯 **Why This Combination?**

- Vite: Fastest development experience
- TypeScript: Catches errors before they reach users
- React Hook Form: Industry standard for complex forms
- All work perfectly together!

---

2. Setting Up with Vite + TypeScript

Prerequisites

You'll need:

- Node.js (version 16 or higher)
- Basic JavaScript/TypeScript knowledge
- A code editor (VS Code recommended)

Creating Your First React Project

Open your terminal and run:

```
npm create vite@latest my-react-app -- --template react-ts
cd my-react-app
npm install
npm run dev
```

Understanding the Project Structure

```
my-react-app/
├── public/           # Static files
├── src/             # Your code lives here
│   ├── App.tsx      # Main app component
│   ├── main.tsx     # Entry point
│   └── index.css    # Global styles
```

```
├── package.json     # Dependencies and scripts
└── tsconfig.json    # TypeScript configuration
```

Key Files Explained

src/main.tsx - The starting point:

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.tsx'

createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <App />
  </StrictMode>,
)
```

src/App.tsx - Your main component:

```
import { useState } from 'react'
import reactLogo from './assets/react.svg'
import viteLogo from '/vite.svg'
import './App.css'

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

  return (
    <>
      <div>
        <a href="https://vite.dev" target="_blank">
          <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
        <a href="https://react.dev" target="_blank">
          <img src={reactLogo} className="logo react" alt="React logo" />
        </a>
      </div>
      <h1>Vite + React</h1>
      <div class="card">
        <button onClick={() => setCount((count) => count + 1)}>
          count is {count}
        </button>
        <p>
          Edit <code>src/App.tsx</code> and save to test HMR
        </p>
      </div>
      <p class="read-the-docs">
        Click on the Vite and React logos to learn more
      </p>
```

```
    </>
  )
}

export default App
```

Exercise: Your First Modification

1. Start your development server: `npm run dev`
2. Open your browser to the URL shown (usually `http://localhost:5173`)
3. Modify the `<h1>` text in `App.tsx`
4. Save the file and watch it update automatically!

Challenge: Change the initial count value to 10 and add a "Reset" button that sets the count back to 0.

---

3. JSX and TSX Basics

What is JSX/TSX?

JSX is JavaScript XML - it lets you write HTML-like code inside JavaScript. TSX is the TypeScript version of JSX.

```
// This looks like HTML, but it's actually JavaScript!
const element = <h1>Hello, World!</h1>
```

JSX Rules

1. Single Parent Element

```
// ❌ Wrong - multiple top-level elements
function BadComponent() {
  return (
    <h1>Title</h1>
    <p>Paragraph</p>
  )
}

// ☑ Correct - wrapped in a single parent
function GoodComponent() {
  return (
    <div>
      <h1>Title</h1>
      <p>Paragraph</p>
    </div>
  )
}
```

```
// ☑ Also correct - using React Fragment
function AlsoGoodComponent() {
  return (
    <>
      <h1>Title</h1>
      <p>Paragraph</p>
    </>
  )
}
```

2. Self-Closing Tags

```
// ✗ Wrong
<input type="text">
<img src="photo.jpg">

// ☑ Correct
<input type="text" />
<img src="photo.jpg" />
```

3. camelCase Attributes

```
// HTML uses kebab-case
<div class="container" onclick="handleClick()">

// JSX uses camelCase
<div className="container" onClick={handleClick}>
```

Embedding JavaScript in JSX

Use curly braces {} to embed JavaScript expressions:

```
function Greeting() {
  const name = "Alice"
  const age = 25

  return (
    <div>
      <h1>Hello, {name}!</h1>
      <p>You are {age} years old.</p>
      <p>Next year you'll be {age + 1}.</p>
    </div>
  )
}
```

TypeScript in TSX

TypeScript helps catch errors in your JSX:

```
interface User {
  name: string
```

```
  age: number
  isActive: boolean
}

function UserCard({ user }: { user: User }) {
  return (
    <div className="user-card">
      <h2>{user.name}</h2>
      <p>Age: {user.age}</p>
      <span className={user.isActive ? 'active' : 'inactive'}>
        {user.isActive ? 'Online' : 'Offline'}
      </span>
    </div>
  )
}
```

Styling in JSX

CSS Classes:

```
// Use className instead of class
<div className="container">
  <button className="btn btn-primary">Click me</button>
</div>
```

Inline Styles:

```
// Style object with camelCase properties
const buttonStyle = {
  backgroundColor: 'blue',
  color: 'white',
  fontSize: '16px',
  border: 'none',
  borderRadius: '4px'
}

<button style={buttonStyle}>Styled Button</button>

// Or inline
<button style={{ backgroundColor: 'red', color: 'white' }}>
  Red Button
</button>
```

Conditional Rendering in JSX

```
function WelcomeMessage({ isLoggedIn, username }: {
  isLoggedIn: boolean
  username?: string
}) {
  return (
    <div>
```

```
      {isLoggedIn ? (
        <h1>Welcome back, {username}!</h1>
      ) : (
        <h1>Please sign in</h1>
      )}
    </div>
  )
}
```

Lists in JSX

```
function TodoList() {
  const todos = ['Buy groceries', 'Walk the dog', 'Write code']

  return (
    <ul>
      {todos.map((todo, index) => (
        <li key={index}>{todo}</li>
      ))}
    </ul>
  )
}
```

Common Pitfalls

❌ Forgetting the key prop in lists

```
// React will warn about missing keys
{items.map(item => <div>{item}</div>)}
```

✅ Always provide unique keys

```
{items.map((item, index) => (
  <div key={item.id || index}>{item}</div>
))}
```

❌ Using class instead of className

```
<div class="container">  // This won't work!
```

✅ Use className

```
<div className="container">  // This works!
```

Exercise: Practice JSX

Create a component that displays information about a book:

```
interface Book {
  title: string
  author: string
  year: number
  isAvailable: boolean
}

// Your task: Create a BookCard component that displays:
// - Book title as a heading
// - Author name
// - Publication year
// - Availability status (Available/Out of Stock) with different colors
// - A "Borrow" button that's disabled if not available
```

Solution:

```
function BookCard({ book }: { book: Book }) {
  return (
    <div className="book-card">
      <h3>{book.title}</h3>
      <p>by {book.author}</p>
      <p>Published: {book.year}</p>
      <p style={{ color: book.isAvailable ? 'green' : 'red' }}>
        {book.isAvailable ? 'Available' : 'Out of Stock'}
      </p>
      <button disabled={!book.isAvailable}>
        Borrow
      </button>
    </div>
  )
}
```

---

4. Creating Components

What are Components?

Components are the building blocks of React applications. Think of them as custom HTML elements that you can reuse throughout your app.

```
Traditional HTML:
<div>
  <h1>Welcome</h1>
  <p>This is a welcome message</p>
</div>
```

```
React Component:
<WelcomeMessage />
```

Component Types

1. Function Components (Recommended)

```
function Welcome() {
  return <h1>Hello, World!</h1>
}

// Or arrow function
const Welcome = () => {
  return <h1>Hello, World!</h1>
}
```

2. Class Components (Legacy, but good to know)

```
import { Component } from 'react'

class Welcome extends Component {
  render() {
    return <h1>Hello, World!</h1>
  }
}
```

💡 Why Function Components?

- Simpler syntax
- Better performance
- Hooks support (we'll learn these later)
- Modern React best practice

Creating Your First Component

Let's build a `UserProfile` component:

```
// src/components/UserProfile.tsx
interface UserProfileProps {
  name: string
  email: string
  avatar: string
}

function UserProfile({ name, email, avatar }: UserProfileProps) {
  return (
    <div className="user-profile">
      <img src={avatar} alt={`${name}'s avatar`} />
      <h2>{name}</h2>
```

```
      <p>{email}</p>
    </div>
  )
}

export default UserProfile
```

Using Components

```
// src/App.tsx
import UserProfile from './components/UserProfile'

function App() {
  return (
    <div>
      <h1>Our Team</h1>
      <UserProfile
        name="Alice Johnson"
        email="alice@example.com"
        avatar="https://example.com/alice.jpg"
      />
      <UserProfile
        name="Bob Smith"
        email="bob@example.com"
        avatar="https://example.com/bob.jpg"
      />
    </div>
  )
}
```

Component File Organization

```
src/
├── components/           # Reusable components
│   ├── UserProfile.tsx
│   ├── Button.tsx
│   └── Header.tsx
├── pages/               # Page components
│   ├── Home.tsx
│   └── About.tsx
├── hooks/               # Custom hooks (later)
├── utils/               # Helper functions
└── App.tsx
```

TypeScript for Components

Props Interface:

```
interface ButtonProps {
  text: string
  onClick: () => void
  disabled?: boolean  // Optional prop
```

```
  variant?: 'primary' | 'secondary' | 'danger'  // Union type
}

function Button({ text, onClick, disabled = false, variant = 'primary' }:
ButtonProps) {
  const className = `btn btn-${variant}`

  return (
    <button
      className={className}
      onClick={onClick}
      disabled={disabled}
    >
      {text}
    </button>
  )
}
```

Children Props:

```
interface CardProps {
  title: string
  children: React.ReactNode  // For nested content
}

function Card({ title, children }: CardProps) {
  return (
    <div className="card">
      <h3 className="card-title">{title}</h3>
      <div className="card-content">
        {children}
      </div>
    </div>
  )
}

// Usage
<Card title="Welcome">
  <p>This is inside the card!</p>
  <button>Click me</button>
</Card>
```

Component Composition

Build complex UIs by combining simple components:

```
// Simple components
function Avatar({ src, alt }: { src: string; alt: string }) {
```

```
  return <img className="avatar" src={src} alt={alt} />
}

function Badge({ text, type }: { text: string; type: 'success' | 'warning' |
'error' }) {
  return <span className={`badge badge-${type}`}>{text}</span>
}

// Composed component
interface UserCardProps {
  user: {
    id: number
    name: string
    email: string
    avatar: string
    status: 'online' | 'away' | 'offline'
  }
}

function UserCard({ user }: UserCardProps) {
  const badgeType = user.status === 'online' ? 'success' :
                    user.status === 'away' ? 'warning' : 'error'

  return (
    <div className="user-card">
      <Avatar src={user.avatar} alt={user.name} />
      <div className="user-info">
        <h4>{user.name}</h4>
        <p>{user.email}</p>
        <Badge text={user.status} type={badgeType} />
      </div>
    </div>
  )
}
```

Common Pitfalls

❌ Component naming (must start with capital letter)

```
function myComponent() {  // Won't work!
  return <div>Hello</div>
}
```

✅ Correct naming

```
function MyComponent() {  // Works!
  return <div>Hello</div>
}
```

❌ Forgetting to export

```
function Button() {
  return <button>Click me</button>
}
// Component can't be imported elsewhere
```

✅ **Always export your components**

```
function Button() {
  return <button>Click me</button>
}

export default Button
```

Pro Tips

🎯 **Component Design Principles**

1. **Single Responsibility**: Each component should do one thing well
2. **Reusability**: Design components to be used in multiple places
3. **Composition over Inheritance**: Build complex UIs by combining simple components
4. **Props Interface**: Always define TypeScript interfaces for props

📁 **File Naming Conventions**

- PascalCase for component files: `UserProfile.tsx`
- Match the component name: `UserProfile` component in `UserProfile.tsx`
- Use descriptive names: `SubmitButton.tsx` instead of `Button.tsx`

Exercise: Build a Product Card

Create a `ProductCard` component with the following requirements:

Interface:

```
interface Product {
  id: number
  name: string
  price: number
  image: string
  inStock: boolean
  category: string
}
```

Features:

- Display product image, name, and price
- Show "In Stock" or "Out of Stock" status
- Include category badge
- Add "Add to Cart" button (disabled if out of stock)

- Use proper TypeScript types

Bonus Challenge: Create a `ProductGrid` component that displays multiple products in a responsive grid layout.

---

5. Props and State

Understanding Props

Props (properties) are how you pass data from parent components to child components. Think of them like function parameters.

```
// Parent Component
function App() {
  return (
    <Greeting
      name="Alice"
      age={25}
      isVip={true}
    />
  )
}

// Child Component
interface GreetingProps {
  name: string
  age: number
  isVip: boolean
}

function Greeting({ name, age, isVip }: GreetingProps) {
  return (
    <div>
      <h1>Hello, {name}!</h1>
      <p>You are {age} years old</p>
      {isVip && <span className="vip-badge">VIP Member</span>}
    </div>
  )
}
```

Props Best Practices

1. Always define TypeScript interfaces:

```
interface ButtonProps {
  text: string
  onClick: () => void
  disabled?: boolean
```

```
  size?: 'small' | 'medium' | 'large'
}
```

2. Use destructuring for cleaner code:

```
// ❌ Less readable
function Button(props: ButtonProps) {
  return (
    <button onClick={props.onClick} disabled={props.disabled}>
      {props.text}
    </button>
  )
}

// ✅ Much cleaner
function Button({ text, onClick, disabled = false }: ButtonProps) {
  return (
    <button onClick={onClick} disabled={disabled}>
      {text}
    </button>
  )
}
```

3. Provide default values:

```
function Avatar({
  src,
  alt,
  size = 'medium',
  rounded = true
}: {
  src: string
  alt: string
  size?: 'small' | 'medium' | 'large'
  rounded?: boolean
}) {
  const sizeClass = `avatar-${size}`
  const roundedClass = rounded ? 'avatar-rounded' : ''

  return (
    <img
      src={src}
      alt={alt}
      className={`avatar ${sizeClass} ${roundedClass}`}
    />
  )
}
```

Understanding State

State is data that can change over time in your component. When state changes, React automatically re-renders the component.

The `useState` Hook:

```
import { useState } from 'react'

function Counter() {
  // useState returns [currentValue, setterFunction]
  const [count, setCount] = useState<number>(0)

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

  const decrement = () => {
    setCount(count - 1)
  }

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={increment}>+</button>
      <button onClick={decrement}>-</button>
    </div>
  )
}
```

State vs Props

```
Props:
- Data passed from parent to child
- Read-only (cannot be modified by child)
- Like function parameters

State:
- Data managed within a component
- Can be modified using setter functions
- Triggers re-render when changed
```

Different Types of State

1. Simple Values:

```
const [name, setName] = useState<string>('')
const [age, setAge] = useState<number>(0)
const [isVisible, setIsVisible] = useState<boolean>(false)
```

2. Objects:

```
interface User {
  name: string
  email: string
  age: number
}

const [user, setUser] = useState<User>({
  name: '',
  email: '',
  age: 0
})

// Updating object state
const updateUserName = (newName: string) => {
  setUser(prevUser => ({
    ...prevUser,  // Spread existing properties
    name: newName // Update only the name
  }))
}
```

3. Arrays:

```
const [todos, setTodos] = useState<string[]>([])

// Adding to array
const addTodo = (newTodo: string) => {
  setTodos(prevTodos => [...prevTodos, newTodo])
}

// Removing from array
const removeTodo = (index: number) => {
  setTodos(prevTodos => prevTodos.filter((_, i) => i !== index))
}
```

State Updates are Async

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

  const handleClick = () => {
    setCount(count + 1)
    console.log(count) // This will log the OLD value!

    // If you need the new value immediately:
    setCount(prevCount => {
      const newCount = prevCount + 1
      console.log(newCount) // This logs the NEW value
      return newCount
    })
  }
```

```
  return <button onClick={handleClick}>Count: {count}</button>
}
```

Lifting State Up

When multiple components need to share state, move it to their common parent:

```
// ❌ Each component has its own state (not synchronized)
function ComponentA() {
  const [value, setValue] = useState('')
  return <input value={value} onChange={e => setValue(e.target.value)} />
}

function ComponentB() {
  const [value, setValue] = useState('')
  return <p>Current value: {value}</p>
}

// ✅ Shared state in parent component
function Parent() {
  const [sharedValue, setSharedValue] = useState('')

  return (
    <div>
      <InputComponent value={sharedValue} onChange={setSharedValue} />
      <DisplayComponent value={sharedValue} />
    </div>
  )
}

function InputComponent({ value, onChange }: {
  value: string
  onChange: (value: string) => void
}) {
  return (
    <input
      value={value}
      onChange={e => onChange(e.target.value)}
    />
  )
}

function DisplayComponent({ value }: { value: string }) {
  return <p>Current value: {value}</p>
}
```

Common Pitfalls

❌ Directly mutating state

```
const [user, setUser] = useState({ name: 'Alice', age: 25 })

// DON'T do this:
user.age = 26  // Direct mutation
setUser(user)  // React won't detect the change
```

✅ Create new objects/arrays

```
// DO this:
setUser(prevUser => ({
  ...prevUser,
  age: 26
}))
```

❌ Using stale state in callbacks

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

useEffect(() => {
  const timer = setInterval(() => {
    setCount(count + 1)  // Always uses initial count (0)
  }, 1000)

  return () => clearInterval(timer)
}, [])
```

✅ Use functional updates

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

useEffect(() => {
  const timer = setInterval(() => {
    setCount(prevCount => prevCount + 1)  // Uses current count
  }, 1000)

  return () => clearInterval(timer)
}, [])
```

Exercise: Shopping Cart Component

Build a shopping cart with the following features:

```
interface CartItem {
  id: number
  name: string
  price: number
  quantity: number
```

```
}

interface Product {
  id: number
  name: string
  price: number
}

// Your tasks:
// 1. Create a ShoppingCart component that manages cart state
// 2. Add products to cart
// 3. Update quantities
// 4. Remove items
// 5. Calculate total price
// 6. Display cart contents
```

Starter Code:

```
function ShoppingCart() {
  const [cartItems, setCartItems] = useState<CartItem[]>([])

  const addToCart = (product: Product) => {
    // Your implementation here
  }

  const updateQuantity = (id: number, newQuantity: number) => {
    // Your implementation here
  }

  const removeFromCart = (id: number) => {
    // Your implementation here
  }

  const getTotalPrice = (): number => {
    // Your implementation here
    return 0
  }

  return (
    <div className="shopping-cart">
      {/* Your JSX here */}
    </div>
  )
}
```

---

6. Event Handling

What are Events?

Events are actions that occur in your application - clicks, form submissions, key presses, mouse movements, etc. React handles events through **event handlers** - functions that run when events occur.

Basic Event Handling

```
function Button() {
  const handleClick = () => {
    console.log('Button was clicked!')
  }

  return (
    <button onClick={handleClick}>
      Click me
    </button>
  )
}
```

Common Event Types

Click Events:

```
function ClickExample() {
  const handleButtonClick = () => {
    alert('Button clicked!')
  }

  const handleDivClick = () => {
    console.log('Div clicked!')
  }

  return (
    <div onClick={handleDivClick}>
      <button onClick={handleButtonClick}>Click me</button>
      <p>Click anywhere in this div</p>
    </div>
  )
}
```

Form Events:

```
function FormExample() {
  const [inputValue, setInputValue] = useState('')

  const handleInputChange = (event: React.ChangeEvent<HTMLInputElement>) => {
    setInputValue(event.target.value)
  }
```

```
  const handleSubmit = (event: React.FormEvent<HTMLFormElement>) => {
    event.preventDefault() // Prevent page refresh
    console.log('Form submitted with:', inputValue)
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={inputValue}
        onChange={handleInputChange}
        placeholder="Enter text"
      />
      <button type="submit">Submit</button>
    </form>
  )
}
```

Keyboard Events:

```
function KeyboardExample() {
  const [message, setMessage] = useState('')

  const handleKeyDown = (event: React.KeyboardEvent<HTMLInputElement>) => {
    if (event.key === 'Enter') {
      setMessage('Enter key pressed!')
    } else if (event.key === 'Escape') {
      setMessage('Escape key pressed!')
    }
  }

  return (
    <div>
      <input
        type="text"
        onKeyDown={handleKeyDown}
        placeholder="Press Enter or Escape"
      />
      <p>{message}</p>
    </div>
  )
}
```

Event Object and TypeScript

React provides type-safe event objects:

```
function EventTypesExample() {
  // Mouse events
```

```
  const handleMouseEvent = (event: React.MouseEvent<HTMLButtonElement>) => {
    console.log('Mouse position:', event.clientX, event.clientY)
    console.log('Button pressed:', event.button) // 0 = left, 1 = middle, 2 =
right
  }

  // Input events
  const handleInputEvent = (event: React.ChangeEvent<HTMLInputElement>) => {
    console.log('Input value:', event.target.value)
    console.log('Input type:', event.target.type)
  }

  // Focus events
  const handleFocusEvent = (event: React.FocusEvent<HTMLInputElement>) => {
    console.log('Input focused')
    event.target.style.backgroundColor = 'lightblue'
  }

  const handleBlurEvent = (event: React.FocusEvent<HTMLInputElement>) => {
    console.log('Input lost focus')
    event.target.style.backgroundColor = 'white'
  }

  return (
    <div>
      <button
        onClick={handleMouseEvent}
        onMouseEnter={() => console.log('Mouse entered')}
        onMouseLeave={() => console.log('Mouse left')}
      >
        Hover and click me
      </button>

      <input
        type="text"
        onChange={handleInputEvent}
        onFocus={handleFocusEvent}
        onBlur={handleBlurEvent}
        placeholder="Focus on me"
      />
    </div>
  )
}
```

Passing Arguments to Event Handlers

Method 1: Arrow Function in JSX

```
function TodoList() {
  const [todos, setTodos] = useState([
```

```
    { id: 1, text: 'Learn React', completed: false },
    { id: 2, text: 'Build an app', completed: false }
  ])

  const toggleTodo = (id: number) => {
    setTodos(prevTodos =>
      prevTodos.map(todo =>
        todo.id === id
          ? { ...todo, completed: !todo.completed }
          : todo
      )
    )
  }

  const deleteTodo = (id: number) => {
    setTodos(prevTodos => prevTodos.filter(todo => todo.id !== id))
  }

  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>
          <span
            style={{
              textDecoration: todo.completed ? 'line-through' : 'none'
            }}
          >
            {todo.text}
          </span>
          {/* Method 1: Arrow function in JSX */}
          <button onClick={() => toggleTodo(todo.id)}>
            {todo.completed ? 'Undo' : 'Complete'}
          </button>
          <button onClick={() => deleteTodo(todo.id)}>
            Delete
          </button>
        </li>
      ))}
    </ul>
  )
}
```

Method 2: Higher-Order Function

```
function TodoList() {
  // ... same state setup

  const createToggleHandler = (id: number) => {
    return () => toggleTodo(id)
  }
```

```
  const createDeleteHandler = (id: number) => {
    return () => deleteTodo(id)
  }

  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>
          <span>{todo.text}</span>
          {/* Method 2: Higher-order function */}
          <button onClick={createToggleHandler(todo.id)}>
            {todo.completed ? 'Undo' : 'Complete'}
          </button>
          <button onClick={createDeleteHandler(todo.id)}>
            Delete
          </button>
        </li>
      ))}
    </ul>
  )
}
```

Event Propagation

Events "bubble up" from child to parent elements:

```
function EventPropagationExample() {
  const handleParentClick = () => {
    console.log('Parent clicked')
  }

  const handleChildClick = (event: React.MouseEvent) => {
    console.log('Child clicked')
    // event.stopPropagation() // Uncomment to stop bubbling
  }

  return (
    <div
      onClick={handleParentClick}
      style={{ padding: '50px', backgroundColor: 'lightgray' }}
    >
      <button
        onClick={handleChildClick}
        style={{ padding: '20px' }}
      >
        Click me (child)
      </button>
      <p>Click anywhere in the gray area (parent)</p>
    </div>
```

```
  )
}
```

Preventing Default Behavior

```
function PreventDefaultExample() {
  const handleLinkClick = (event: React.MouseEvent<HTMLAnchorElement>) => {
    event.preventDefault() // Prevent navigation
    console.log('Link clicked, but navigation prevented')
  }

  const handleFormSubmit = (event: React.FormEvent<HTMLFormElement>) => {
    event.preventDefault() // Prevent page refresh
    console.log('Form submitted, but page refresh prevented')
  }

  return (
    <div>
      <a href="https://example.com" onClick={handleLinkClick}>
        Click me (won't navigate)
      </a>

      <form onSubmit={handleFormSubmit}>
        <input type="text" placeholder="Enter something" />
        <button type="submit">Submit (won't refresh)</button>
      </form>
    </div>
  )
}
```

Handling Multiple Inputs

When dealing with forms that have multiple inputs:

```
interface FormData {
  firstName: string
  lastName: string
  email: string
  age: number
}

function MultiInputForm() {
  const [formData, setFormData] = useState<FormData>({
    firstName: '',
    lastName: '',
    email: '',
    age: 0
  })

  // Generic handler for all inputs
  const handleInputChange = (event: React.ChangeEvent<HTMLInputElement>) => {
```

```
  const { name, value, type } = event.target

  setFormData(prevData => ({
    ...prevData,
    [name]: type === 'number' ? Number(value) : value
  }))
}

const handleSubmit = (event: React.FormEvent) => {
  event.preventDefault()
  console.log('Form data:', formData)
}

return (
  <form onSubmit={handleSubmit}>
    <input
      type="text"
      name="firstName"
      value={formData.firstName}
      onChange={handleInputChange}
      placeholder="First Name"
    />

    <input
      type="text"
      name="lastName"
      value={formData.lastName}
      onChange={handleInputChange}
      placeholder="Last Name"
    />

    <input
      type="email"
      name="email"
      value={formData.email}
      onChange={handleInputChange}
      placeholder="Email"
    />

    <input
      type="number"
      name="age"
      value={formData.age}
      onChange={handleInputChange}
      placeholder="Age"
    />

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

```
  )
}
```

Event Handler Performance

❌ **Creating functions in render (causes unnecessary re-renders):**

```
function BadExample({ items }: { items: string[] }) {
  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>
          {/* New function created on every render */}
          <button onClick={() => console.log(item)}>
            {item}
          </button>
        </li>
      ))}
    </ul>
  )
}
```

✅ **Optimize with useCallback (we'll learn this later):**

```
import { useCallback } from 'react'

function GoodExample({ items }: { items: string[] }) {
  // Memoized function - only recreated if dependencies change
  const createClickHandler = useCallback((item: string) => {
    return () => console.log(item)
  }, [])

  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>
          <button onClick={createClickHandler(item)}>
            {item}
          </button>
        </li>
      ))}
    </ul>
  )
}
```

Common Pitfalls

❌ **Calling function immediately instead of passing reference:**

```
function BadButton() {
  const handleClick = () => {
    console.log('Clicked!')
  }

  return (
    <button onClick={handleClick()}>  {/* Wrong! Calls immediately */}
      Click me
    </button>
  )
}
```

✅ **Pass function reference:**

```
function GoodButton() {
  const handleClick = () => {
    console.log('Clicked!')
  }

  return (
    <button onClick={handleClick}>  {/* Correct! Passes reference */}
      Click me
    </button>
  )
}
```

❌ **Forgetting event types in TypeScript:**

```
function BadHandler() {
  const handleChange = (event) => {  // TypeScript error!
    console.log(event.target.value)
  }

  return <input onChange={handleChange} />
}
```

✅ **Always type your events:**

```
function GoodHandler() {
  const handleChange = (event: React.ChangeEvent<HTMLInputElement>) => {
    console.log(event.target.value)
  }

  return <input onChange={handleChange} />
}
```

Pro Tips

⚡ **Performance Tip**: For lists with many items, consider using event delegation or memoization to avoid creating too many event handlers.

🎯 **Naming Convention**: Prefix event handler functions with "handle" followed by the event type: `handleClick`, `handleSubmit`, `handleInputChange`.

🔍 **Debugging Events**: Use `console.log(event)` to inspect event objects and understand what data is available.

Exercise: Interactive Counter with History

Build an enhanced counter with the following features:

```
interface CounterAction {
  type: 'increment' | 'decrement' | 'reset'
  timestamp: Date
  previousValue: number
  newValue: number
}

// Requirements:
// 1. Counter with increment, decrement, and reset buttons
// 2. Track history of all actions
// 3. Undo last action functionality
// 4. Keyboard shortcuts (+ for increment, - for decrement, r for reset)
// 5. Display action history with timestamps
// 6. Clear history button
```

Starter Code:

```
function EnhancedCounter() {
  const [count, setCount] = useState(0)
  const [history, setHistory] = useState<CounterAction[]>([])

  const handleIncrement = () => {
    // Your implementation
  }

  const handleDecrement = () => {
    // Your implementation
  }

  const handleReset = () => {
    // Your implementation
  }

  const handleUndo = () => {
    // Your implementation
  }

  const handleKeyDown = (event: React.KeyboardEvent) => {
    // Your implementation for keyboard shortcuts
  }
```

```
  return (
    <div onKeyDown={handleKeyDown} tabIndex={0}>
      {/* Your JSX implementation */}
    </div>
  )
}
```

---

7. Conditional Rendering

What is Conditional Rendering?

Conditional rendering means showing different content based on certain conditions. It's like an if-else statement for your UI.

```
function WelcomeMessage({ isLoggedIn, username }: {
  isLoggedIn: boolean
  username?: string
}) {
  if (isLoggedIn) {
    return <h1>Welcome back, {username}!</h1>
  } else {
    return <h1>Please sign in</h1>
  }
}
```

Conditional Rendering Techniques

1. If-Else Statements:

```
function UserDashboard({ user }: { user: User | null }) {
  if (!user) {
    return (
      <div className="loading">
        <p>Loading user data...</p>
      </div>
    )
  }

  if (user.role === 'admin') {
    return (
      <div className="admin-dashboard">
        <h1>Admin Panel</h1>
        <AdminControls />
        <UserManagement />
      </div>
    )
  }
```

```
  return (
    <div className="user-dashboard">
      <h1>Welcome, {user.name}</h1>
      <UserProfile user={user} />
    </div>
  )
}
```

2. Ternary Operator (`condition ? true : false`):

```
function StatusBadge({ isOnline }: { isOnline: boolean }) {
  return (
    <span className={isOnline ? 'status-online' : 'status-offline'}>
      {isOnline ? '🟢 Online' : '🔴 Offline'}
    </span>
  )
}

function ShoppingCart({ items }: { items: CartItem[] }) {
  return (
    <div className="cart">
      <h2>Shopping Cart</h2>
      {items.length > 0 ? (
        <div>
          <CartItems items={items} />
          <CheckoutButton />
        </div>
      ) : (
        <p>Your cart is empty</p>
      )}
    </div>
  )
}
```

3. Logical AND (&&) Operator:

```
function Notifications({
  notifications,
  showNotifications
}: {
  notifications: string[]
  showNotifications: boolean
}) {
  return (
    <div>
      <h2>Dashboard</h2>

      {/* Only show if condition is true */}
      {showNotifications && (
```

```
        <div className="notifications">
          <h3>Notifications ({notifications.length})</h3>
          {notifications.map((notification, index) => (
            <div key={index} className="notification">
              {notification}
            </div>
          ))}
        </div>
      )}

      {/* Show badge only if there are notifications */}
      {notifications.length > 0 && (
        <span className="notification-badge">
          {notifications.length}
        </span>
      )}
    </div>
  )
}
```

4. Logical OR (||) for Default Values:

```
function UserProfile({ user }: { user: User }) {
  return (
    <div className="profile">
      <img
        src={user.avatar || '/default-avatar.png'}
        alt="Profile"
      />
      <h2>{user.name || 'Anonymous User'}</h2>
      <p>{user.bio || 'No bio available'}</p>
    </div>
  )
}
```

Complex Conditional Logic

Multiple Conditions:

```
interface WeatherDisplayProps {
  temperature: number
  isRaining: boolean
  windSpeed: number
}

function WeatherDisplay({ temperature, isRaining, windSpeed }:
WeatherDisplayProps) {
  const getWeatherIcon = (): string => {
    if (isRaining) return '🌧️'
    if (temperature > 30) return '🔥'
```

```
    if (temperature < 0) return '❄️'
    if (windSpeed > 20) return '💨'
    return '☁️'
  }

  const getWeatherDescription = (): string => {
    if (isRaining && temperature < 5) return 'Cold and rainy'
    if (isRaining && temperature > 25) return 'Warm rain'
    if (temperature > 35) return 'Very hot'
    if (temperature < -5) return 'Freezing'
    return 'Pleasant weather'
  }

  const getClothingAdvice = (): string => {
    if (isRaining) return 'Bring an umbrella!'
    if (temperature > 30) return 'Wear light clothing'
    if (temperature < 10) return 'Dress warmly'
    return 'Comfortable clothing recommended'
  }

  return (
    <div className="weather-display">
      <div className="weather-icon">{getWeatherIcon()}</div>
      <div className="temperature">{temperature}°C</div>
      <div className="description">{getWeatherDescription()}</div>
      <div className="advice">{getClothingAdvice()}</div>

      {/* Conditional styling */}
      <div
        className="weather-background"
        style={{
          backgroundColor:
            temperature > 30 ? '#ffeb3b' :
            temperature < 0 ? '#2196f3' :
            isRaining ? '#607d8b' : '#4caf50'
        }}
      />
    </div>
  )
}
```

Switch-like Logic with Objects:

```
type UserRole = 'admin' | 'moderator' | 'user' | 'guest'

interface User {
  name: string
  role: UserRole
  permissions: string[]
```

```
}

function UserInterface({ user }: { user: User }) {
  // Define what each role can see
  const roleComponents = {
    admin: (
      <div className="admin-interface">
        <AdminPanel />
        <UserManagement />
        <SystemSettings />
      </div>
    ),
    moderator: (
      <div className="moderator-interface">
        <ContentModeration />
        <UserReports />
      </div>
    ),
    user: (
      <div className="user-interface">
        <UserProfile user={user} />
        <UserContent />
      </div>
    ),
    guest: (
      <div className="guest-interface">
        <LoginPrompt />
        <PublicContent />
      </div>
    )
  }

  return (
    <div className="app-interface">
      <Header user={user} />
      {roleComponents[user.role]}
    </div>
  )
}
```

Conditional Classes and Styles

Dynamic Class Names:

```
function Button({
  variant,
  size,
  disabled,
  active
}: {
```

```
  variant: 'primary' | 'secondary' | 'danger'
  size: 'small' | 'medium' | 'large'
  disabled?: boolean
  active?: boolean
}) {
  // Build class name dynamically
  const className = [
    'btn',
    `btn-${variant}`,
    `btn-${size}`,
    disabled && 'btn-disabled',
    active && 'btn-active'
  ]
    .filter(Boolean) // Remove falsy values
    .join(' ')

  return (
    <button
      className={className}
      disabled={disabled}
    >
      Button Text
    </button>
  )
}

// Alternative using template literals
function AlternativeButton({ variant, disabled }: {
  variant: 'primary' | 'secondary'
  disabled?: boolean
}) {
  return (
    <button
      className={`
        btn
        btn-${variant}
        ${disabled ? 'btn-disabled' : ''}
      `.trim()}
      disabled={disabled}
    >
      Button Text
    </button>
  )
}
```

Dynamic Inline Styles:

```
function ProgressBar({
  progress,
  color,
```

```
  animated
}: {
  progress: number
  color?: string
  animated?: boolean
}) {
  return (
    <div className="progress-container">
      <div
        className={`progress-bar ${animated ? 'animated' : ''}`}
        style={{
          width: `${Math.min(100, Math.max(0, progress))}%`,
          backgroundColor: color || '#007bff',
          transition: animated ? 'width 0.3s ease' : 'none'
        }}
      />
      <span className="progress-text">
        {Math.round(progress)}%
      </span>
    </div>
  )
}
```

Handling Loading and Error States

```
interface ApiState<T> {
  data: T | null
  loading: boolean
  error: string | null
}

function DataDisplay<T>({
  apiState,
  renderData
}: {
  apiState: ApiState<T>
  renderData: (data: T) => React.ReactNode
}) {
  const { data, loading, error } = apiState

  // Error state
  if (error) {
    return (
      <div className="error-state">
        <h3>❌ Something went wrong</h3>
        <p>{error}</p>
        <button onClick={() => window.location.reload()}>
          Try Again
        </button>
      </div>
```

```
    )
  }

  // Loading state
  if (loading) {
    return (
      <div className="loading-state">
        <div className="spinner" />
        <p>Loading...</p>
      </div>
    )
  }

  // Empty state
  if (!data) {
    return (
      <div className="empty-state">
        <h3>📭 No data available</h3>
        <p>There's nothing to show right now.</p>
      </div>
    )
  }

  // Success state
  return (
    <div className="success-state">
      {renderData(data)}
    </div>
  )
}

// Usage
function UserList() {
  const [apiState, setApiState] = useState<ApiState<User[]>>({
    data: null,
    loading: true,
    error: null
  })

  return (
    <DataDisplay
      apiState={apiState}
      renderData={(users) => (
        <div>
          <h2>Users ({users.length})</h2>
          {users.map(user => (
            <UserCard key={user.id} user={user} />
          ))}
        </div>
```

```
      )}
    />
  )
}
```

Common Pitfalls

❌ Forgetting falsy values can break rendering:

```
function BadExample({ count }: { count: number }) {
  return (
    <div>
      {/* If count is 0, this renders "0" instead of nothing */}
      {count && <p>Count: {count}</p>}
    </div>
  )
}
```

✅ Be explicit with boolean conditions:

```
function GoodExample({ count }: { count: number }) {
  return (
    <div>
      {/* Explicitly check for > 0 */}
      {count > 0 && <p>Count: {count}</p>}
      {/* Or use ternary for more control */}
      {count > 0 ? <p>Count: {count}</p> : null}
    </div>
  )
}
```

❌ Returning undefined instead of null:

```
function BadComponent({ show }: { show: boolean }) {
  if (!show) {
    return undefined // This can cause issues
  }
  return <div>Content</div>
}
```

✅ Always return null for "render nothing":

```
function GoodComponent({ show }: { show: boolean }) {
  if (!show) {
    return null // This is correct
  }
  return <div>Content</div>
}
```

Pro Tips

🎯 **Readability Tip**: For complex conditions, extract them into well-named variables:

```
const isUserAdmin = user.role === 'admin'
const hasPermissions = user.permissions.length > 0
const canAccessPanel = isUserAdmin && hasPermissions

return (
  <div>
    {canAccessPanel && <AdminPanel />}
  </div>
)
```

⚡ **Performance Tip**: For expensive components, use conditional rendering to avoid rendering them when not needed, rather than hiding them with CSS.

Exercise: Feature-Rich Todo App

Build a todo application with advanced conditional rendering:

```
interface Todo {
  id: number
  text: string
  completed: boolean
  priority: 'low' | 'medium' | 'high'
  dueDate?: Date
  category: string
}

interface TodoAppState {
  todos: Todo[]
  filter: 'all' | 'active' | 'completed'
  sortBy: 'dueDate' | 'priority' | 'alphabetical'
  searchTerm: string
  showCategories: boolean
}

// Requirements:
// 1. Show different views based on filter (all/active/completed)
// 2. Display todos with priority-based styling
// 3. Show overdue todos with special styling
// 4. Empty states for different filters
// 5. Search functionality with "no results" state
// 6. Category grouping (toggle on/off)
// 7. Priority badges with different colors
// 8. Due date warnings (due today, overdue)
```

Challenge: Implement all the conditional rendering logic while keeping the code readable and maintainable.

Chapter 8: Lists and Keys

Understanding Lists in React

When building web applications, you'll frequently need to display lists of data - whether it's a list of users, products, or blog posts. React provides powerful ways to render these lists dynamically.

Rendering Lists with map()

The most common way to render lists in React is using JavaScript's `map()` method:

```
interface User {
  id: number;
  name: string;
  email: string;
}

const UserList: React.FC = () => {
  const users: User[] = [
    { id: 1, name: "Alice Johnson", email: "alice@example.com" },
    { id: 2, name: "Bob Smith", email: "bob@example.com" },
    { id: 3, name: "Carol Davis", email: "carol@example.com" }
  ];

  return (
    <div>
      <h2>User List</h2>
      <ul>
        {users.map((user) => (
          <li key={user.id}>
            <strong>{user.name}</strong> - {user.email}
          </li>
        ))}
      </ul>
    </div>
  );
};
```

The Importance of Keys

Notice the `key` prop in the example above. Keys help React identify which items have changed, been added, or removed. This is crucial for performance and preventing bugs.

❌ *Common Pitfall: Using Array Index as Key*

```
// DON'T do this when the list can change
{users.map((user, index) => (
  <li key={index}>{user.name}</li>
))}
```

☑ *Best Practice: Use Unique, Stable IDs*

```
// DO this - use a unique, stable identifier
{users.map((user) => (
  <li key={user.id}>{user.name}</li>
))}
```

Creating Reusable List Components

Let's create a more complex example with reusable components:

```
interface Product {
  id: number;
  name: string;
  price: number;
  category: string;
  inStock: boolean;
}

interface ProductCardProps {
  product: Product;
  onAddToCart: (product: Product) => void;
}

const ProductCard: React.FC<ProductCardProps> = ({ product, onAddToCart }) =>
{
  return (
    <div className="product-card">
      <h3>{product.name}</h3>
      <p>Price: ${product.price.toFixed(2)}</p>
      <p>Category: {product.category}</p>
      <p>Status: {product.inStock ? "In Stock" : "Out of Stock"}</p>
      <button
        onClick={() => onAddToCart(product)}
        disabled={!product.inStock}
      >
        Add to Cart
      </button>
    </div>
  );
};

interface ProductListProps {
  products: Product[];
}
```

```
const ProductList: React.FC<ProductListProps> = ({ products }) => {
  const handleAddToCart = (product: Product) => {
    console.log(`Added ${product.name} to cart`);
    // In a real app, you'd update cart state here
  };

  return (
    <div className="product-list">
      <h2>Products ({products.length})</h2>
      {products.length === 0 ? (
        <p>No products available.</p>
      ) : (
        <div className="products-grid">
          {products.map((product) => (
            <ProductCard
              key={product.id}
              product={product}
              onAddToCart={handleAddToCart}
            />
          ))}
        </div>
      )}
    </div>
  );
};
```

Filtering and Sorting Lists

Often, you'll want to filter or sort your lists before rendering:

```
const ProductCatalog: React.FC = () => {
  const [products] = useState<Product[]>([
    { id: 1, name: "Laptop", price: 999, category: "Electronics", inStock:
true },
    { id: 2, name: "Desk Chair", price: 199, category: "Furniture", inStock:
false },
    { id: 3, name: "Phone", price: 699, category: "Electronics", inStock:
true },
    // ... more products
  ]);

  const [filter, setFilter] = useState<string>("all");
  const [sortBy, setSortBy] = useState<"name" | "price">("name");

  // Filter products
  const filteredProducts = products.filter((product) => {
    if (filter === "inStock") return product.inStock;
    if (filter === "outOfStock") return !product.inStock;
    return true; // "all"
```

```
  });

  // Sort products
  const sortedProducts = [...filteredProducts].sort((a, b) => {
    if (sortBy === "price") {
      return a.price - b.price;
    }
    return a.name.localeCompare(b.name);
  });

  return (
    <div>
      <div className="controls">
        <select
          value={filter}
          onChange={(e) => setFilter(e.target.value)}
        >
          <option value="all">All Products</option>
          <option value="inStock">In Stock</option>
          <option value="outOfStock">Out of Stock</option>
        </select>

        <select
          value={sortBy}
          onChange={(e) => setSortBy(e.target.value as "name" | "price")}
        >
          <option value="name">Sort by Name</option>
          <option value="price">Sort by Price</option>
        </select>
      </div>

      <ProductList products={sortedProducts} />
    </div>
  );
};
```

💡 Pro Tips

1. **Always use keys**: Even if your list doesn't change, keys help React optimize rendering
2. **Keep transformations simple**: Don't perform heavy calculations in render - use `useMemo` for expensive operations
3. **Handle empty states**: Always consider what to show when your list is empty

🚨 Common Pitfalls

1. **Mutating arrays directly**: Always create new arrays when filtering/sorting
2. **Using non-unique keys**: This can cause rendering issues and lost component state

3. **Forgetting to handle loading states**: Always consider what to show while data is loading

Exercise: Todo List with Filtering

Create a todo list component that allows you to:

1. Display a list of todos
2. Filter todos by completion status (all, completed, pending)
3. Each todo should have a unique ID and proper key

```
interface Todo {
  id: number;
  text: string;
  completed: boolean;
  createdAt: Date;
}

// Your implementation here
const TodoApp: React.FC = () => {
  // Implement the todo list with filtering
};
```

Chapter 9: Hooks and Lifecycle with useEffect

Understanding Component Lifecycle

In React functional components, we use the `useEffect` hook to handle side effects and lifecycle events. This replaces the lifecycle methods from class components.

Basic useEffect Usage

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

const Timer: React.FC = () => {
  const [seconds, setSeconds] = useState<number>(0);

  // Effect runs after every render
  useEffect(() => {
    console.log('Component rendered, seconds:', seconds);
  });

  // Effect runs only once (on mount)
  useEffect(() => {
    console.log('Component mounted');
  }, []); // Empty dependency array

  // Effect runs when seconds changes
  useEffect(() => {
```

```
    document.title = `Timer: ${seconds}s`;
  }, [seconds]); // Dependency array with seconds

  return (
    <div>
      <h2>Timer: {seconds} seconds</h2>
      <button onClick={() => setSeconds(seconds + 1)}>
        Increment
      </button>
    </div>
  );
};
```

Cleanup with useEffect

Some effects need cleanup to prevent memory leaks:

```
const WindowSize: React.FC = () => {
  const [windowSize, setWindowSize] = useState<{
    width: number;
    height: number;
  }>({
    width: window.innerWidth,
    height: window.innerHeight,
  });

  useEffect(() => {
    const handleResize = () => {
      setWindowSize({
        width: window.innerWidth,
        height: window.innerHeight,
      });
    };

    // Add event listener
    window.addEventListener('resize', handleResize);

    // Cleanup function - runs when component unmounts
    // or before the effect runs again
    return () => {
      window.removeEventListener('resize', handleResize);
    };
  }, []); // Empty dependency array - effect runs once

  return (
    <div>
      <h2>Window Size</h2>
      <p>Width: {windowSize.width}px</p>
      <p>Height: {windowSize.height}px</p>
    </div>
```

```
  );
};
```

Advanced useEffect Patterns

Conditional Effects

```
const UserProfile: React.FC<{ userId: number | null }> = ({ userId }) => {
  const [user, setUser] = useState<User | null>(null);
  const [loading, setLoading] = useState<boolean>(false);

  useEffect(() => {
    // Only run effect if userId exists
    if (!userId) {
      setUser(null);
      return;
    }

    setLoading(true);

    // Simulated API call
    const fetchUser = async () => {
      try {
        // const response = await fetch(`/api/users/${userId}`);
        // const userData = await response.json();

        // Simulated delay
        setTimeout(() => {
          setUser({
            id: userId,
            name: `User ${userId}`,
            email: `user${userId}@example.com`
          });
          setLoading(false);
        }, 1000);
      } catch (error) {
        console.error('Failed to fetch user:', error);
        setLoading(false);
      }
    };

    fetchUser();
  }, [userId]); // Effect runs when userId changes

  if (loading) return <div>Loading user...</div>;
  if (!user) return <div>No user selected</div>;

  return (
    <div>
      <h2>{user.name}</h2>
```

```
      <p>{user.email}</p>
    </div>
  );
};
```

Multiple Effects for Separation of Concerns

```
const Dashboard: React.FC = () => {
  const [user, setUser] = useState<User | null>(null);
  const [notifications, setNotifications] = useState<Notification[]>([]);
  const [onlineUsers, setOnlineUsers] = useState<number>(0);

  // Effect for fetching user data
  useEffect(() => {
    const fetchUser = async () => {
      // Fetch user logic
    };
    fetchUser();
  }, []);

  // Effect for real-time notifications
  useEffect(() => {
    const ws = new WebSocket('ws://localhost:8080/notifications');

    ws.onmessage = (event) => {
      const notification = JSON.parse(event.data);
      setNotifications(prev => [...prev, notification]);
    };

    return () => ws.close();
  }, []);

  // Effect for tracking online users
  useEffect(() => {
    const interval = setInterval(() => {
      // Fetch online users count
      setOnlineUsers(Math.floor(Math.random() * 100));
    }, 30000); // Every 30 seconds

    return () => clearInterval(interval);
  }, []);

  return (
    <div>
      <h1>Dashboard</h1>
      <p>Online users: {onlineUsers}</p>
      <p>Notifications: {notifications.length}</p>
    </div>
  );
};
```

💡 Pro Tips

1. **Separate concerns**: Use multiple `useEffect` hooks for different concerns
2. **Always clean up**: Remove event listeners, cancel API calls, clear timers
3. **Be specific with dependencies**: Include all values from component scope that are used inside the effect

🚨 Common Pitfalls

1. **Missing dependencies**: This can cause stale closures and bugs
2. **Forgetting cleanup**: Memory leaks from event listeners or timers
3. **Infinite loops**: Caused by missing or incorrect dependency arrays

```
// ❌ DON'T - Missing dependency
const BadExample: React.FC = () => {
  const [count, setCount] = useState(0);
  const [name, setName] = useState('');

  useEffect(() => {
    document.title = `${name}: ${count}`;
  }, [count]); // Missing 'name' dependency!

  // ... rest of component
};

// ☑ DO - Include all dependencies
const GoodExample: React.FC = () => {
  const [count, setCount] = useState(0);
  const [name, setName] = useState('');

  useEffect(() => {
    document.title = `${name}: ${count}`;
  }, [count, name]); // All dependencies included

  // ... rest of component
};
```

Exercise: Auto-Save Feature

Create a component that auto-saves user input after they stop typing for 2 seconds:

```
const AutoSaveEditor: React.FC = () => {
  const [content, setContent] = useState<string>('');
  const [lastSaved, setLastSaved] = useState<Date | null>(null);
  const [saving, setSaving] = useState<boolean>(false);

  // Implement auto-save logic using useEffect
  // Hint: Use setTimeout and clearTimeout

  return (
```

```
    <div>
      <textarea
        value={content}
        onChange={(e) => setContent(e.target.value)}
        placeholder="Start typing..."
      />
      <div>
        {saving && <span>Saving...</span>}
        {lastSaved && <span>Last saved:
{lastSaved.toLocaleTimeString()}</span>}
      </div>
    </div>
  );
};
```

Chapter 10: Fetching Data

Understanding Asynchronous Operations in React

Data fetching is a crucial part of most web applications. In React, we typically fetch data using the `useEffect` hook combined with the Fetch API or libraries like Axios.

Basic Data Fetching Pattern

```
interface Post {
  id: number;
  title: string;
  body: string;
  userId: number;
}

const PostList: React.FC = () => {
  const [posts, setPosts] = useState<Post[]>([]);
  const [loading, setLoading] = useState<boolean>(true);
  const [error, setError] = useState<string | null>(null);

  useEffect(() => {
    const fetchPosts = async () => {
      try {
        setLoading(true);
        setError(null);

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

        if (!response.ok) {
          throw new Error(`HTTP error! status: ${response.status}`);
        }
```

```
        const data: Post[] = await response.json();
        setPosts(data.slice(0, 10)); // Only show first 10 posts
      } catch (err) {
        setError(err instanceof Error ? err.message : 'An error occurred');
      } finally {
        setLoading(false);
      }
    };

    fetchPosts();
  }, []);

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

  return (
    <div>
      <h2>Latest Posts</h2>
      {posts.map(post => (
        <article key={post.id} className="post">
          <h3>{post.title}</h3>
          <p>{post.body}</p>
        </article>
      ))}
    </div>
  );
};
```

Creating a Custom Hook for Data Fetching

Let's create a reusable custom hook:

```
interface UseApiResponse<T> {
  data: T | null;
  loading: boolean;
  error: string | null;
  refetch: () => void;
}

function useApi<T>(url: string): UseApiResponse<T> {
  const [data, setData] = useState<T | null>(null);
  const [loading, setLoading] = useState<boolean>(true);
  const [error, setError] = useState<string | null>(null);

  const fetchData = async () => {
    try {
      setLoading(true);
      setError(null);
```

```
      const response = await fetch(url);

      if (!response.ok) {
        throw new Error(`HTTP error! status: ${response.status}`);
      }

      const result: T = await response.json();
      setData(result);
    } catch (err) {
      setError(err instanceof Error ? err.message : 'An error occurred');
    } finally {
      setLoading(false);
    }
  };

  useEffect(() => {
    fetchData();
  }, [url]);

  return {
    data,
    loading,
    error,
    refetch: fetchData
  };
}

// Usage
const UserProfile: React.FC<{ userId: number }> = ({ userId }) => {
  const { data: user, loading, error, refetch } = useApi<User>(
    `https://jsonplaceholder.typicode.com/users/${userId}`
  );

  if (loading) return <div>Loading user...</div>;
  if (error) return <div>Error: {error} <button
onClick={refetch}>Retry</button></div>;
  if (!user) return <div>User not found</div>;

  return (
    <div>
      <h2>{user.name}</h2>
      <p>Email: {user.email}</p>
      <p>Phone: {user.phone}</p>
      <button onClick={refetch}>Refresh</button>
    </div>
  );
};
```

Handling Different Data States

A robust data fetching component should handle various states:

```
interface WeatherData {
  location: string;
  temperature: number;
  description: string;
  humidity: number;
}

const WeatherWidget: React.FC = () => {
  const [weather, setWeather] = useState<WeatherData | null>(null);
  const [loading, setLoading] = useState<boolean>(false);
  const [error, setError] = useState<string | null>(null);
  const [city, setCity] = useState<string>('London');

  const fetchWeather = async (cityName: string) => {
    try {
      setLoading(true);
      setError(null);

      // Simulated API call
      const response = await new Promise<WeatherData>((resolve, reject) => {
        setTimeout(() => {
          if (cityName.toLowerCase() === 'invalid') {
            reject(new Error('City not found'));
          } else {
            resolve({
              location: cityName,
              temperature: Math.round(Math.random() * 30 + 5),
              description: 'Partly cloudy',
              humidity: Math.round(Math.random() * 100)
            });
          }
        }, 1000);
      });

      setWeather(response);
    } catch (err) {
      setError(err instanceof Error ? err.message : 'Failed to fetch
weather');
      setWeather(null);
    } finally {
      setLoading(false);
    }
  };

  useEffect(() => {
    fetchWeather(city);
```

```
}, [city]);

const handleSearch = (e: React.FormEvent<HTMLFormElement>) => {
  e.preventDefault();
  const formData = new FormData(e.currentTarget);
  const newCity = formData.get('city') as string;
  if (newCity.trim()) {
    setCity(newCity.trim());
  }
};

return (
  <div className="weather-widget">
    <form onSubmit={handleSearch}>
      <input
        name="city"
        type="text"
        placeholder="Enter city name"
        defaultValue={city}
      />
      <button type="submit" disabled={loading}>
        Search
      </button>
    </form>

    {loading && (
      <div className="loading">
        <div className="spinner"></div>
        <p>Fetching weather data...</p>
      </div>
    )}

    {error && (
      <div className="error">
        <p>❌ {error}</p>
        <button onClick={() => fetchWeather(city)}>
          Try Again
        </button>
      </div>
    )}

    {weather && !loading && (
      <div className="weather-info">
        <h3>🌍 {weather.location}</h3>
        <div className="temperature">{weather.temperature}°C</div>
        <p>{weather.description}</p>
        <p>Humidity: {weather.humidity}%</p>
      </div>
    )}
```

```
      {!weather && !loading && !error && (
        <div className="no-data">
          <p>Enter a city name to get weather information</p>
        </div>
      )}
    </div>
  );
};
```

Advanced Pattern: Data Fetching with Caching

```
interface CacheEntry<T> {
  data: T;
  timestamp: number;
}

class DataCache {
  private cache = new Map<string, CacheEntry<any>>();
  private readonly TTL = 5 * 60 * 1000; // 5 minutes

  get<T>(key: string): T | null {
    const entry = this.cache.get(key);
    if (!entry) return null;

    if (Date.now() - entry.timestamp > this.TTL) {
      this.cache.delete(key);
      return null;
    }

    return entry.data;
  }

  set<T>(key: string, data: T): void {
    this.cache.set(key, {
      data,
      timestamp: Date.now()
    });
  }
}

const cache = new DataCache();

function useApiWithCache<T>(url: string): UseApiResponse<T> {
  const [data, setData] = useState<T | null>(null);
  const [loading, setLoading] = useState<boolean>(true);
  const [error, setError] = useState<string | null>(null);

  const fetchData = async () => {
    try {
```

```
      setLoading(true);
      setError(null);

      // Check cache first
      const cachedData = cache.get<T>(url);
      if (cachedData) {
        setData(cachedData);
        setLoading(false);
        return;
      }

      const response = await fetch(url);

      if (!response.ok) {
        throw new Error(`HTTP error! status: ${response.status}`);
      }

      const result: T = await response.json();

      // Cache the result
      cache.set(url, result);
      setData(result);
    } catch (err) {
      setError(err instanceof Error ? err.message : 'An error occurred');
    } finally {
      setLoading(false);
    }
  };

  useEffect(() => {
    fetchData();
  }, [url]);

  return {
    data,
    loading,
    error,
    refetch: fetchData
  };
}
```

💡 Pro Tips

1. **Always handle loading and error states**: Users need feedback during async operations
2. **Use AbortController**: Cancel requests when components unmount to prevent memory leaks
3. **Implement retry logic**: Network requests can fail temporarily
4. **Consider caching**: Avoid unnecessary API calls for the same data

🚨 Common Pitfalls

1. **Race conditions**: Multiple async calls can complete out of order
2. **Memory leaks**: Setting state after component unmounts
3. **Not handling network errors**: Always wrap fetch calls in try-catch
4. **Infinite re-renders**: Incorrect dependency arrays in useEffect

Exercise: GitHub User Search

Create a component that searches for GitHub users and displays their information:

```
interface GitHubUser {
  id: number;
  login: string;
  name: string;
  avatar_url: string;
  public_repos: number;
  followers: number;
  following: number;
}

const GitHubUserSearch: React.FC = () => {
  // Implement GitHub user search using the GitHub API
  // https://api.github.com/users/{username}

  // Features to implement:
  // 1. Search input with debounced search (wait 500ms after user stops
typing)
  // 2. Loading state while searching
  // 3. Error handling for users not found
  // 4. Display user information when found
  // 5. Handle empty states appropriately

  return (
    <div>
      {/* Your implementation here */}
    </div>
  );
};
```

Chapter 11: Handling Form Inputs and Two-Way Binding

Understanding Controlled vs Uncontrolled Components

In React, there are two ways to handle form inputs:

1. **Controlled Components**: React controls the input value through state

2. **Uncontrolled Components**: The DOM manages the input value

Controlled Components - The React Way

With controlled components, React state is the "single source of truth":

```
const BasicForm: React.FC = () => {
  const [name, setName] = useState<string>('');
  const [email, setEmail] = useState<string>('');
  const [age, setAge] = useState<number>(0);

  const handleSubmit = (e: React.FormEvent<HTMLFormElement>) => {
    e.preventDefault();
    console.log('Form data:', { name, email, age });
  };

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label htmlFor="name">Name:</label>
        <input
          type="text"
          id="name"
          value={name}
          onChange={(e) => setName(e.target.value)}
          placeholder="Enter your name"
        />
      </div>

      <div>
        <label htmlFor="email">Email:</label>
        <input
          type="email"
          id="email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
          placeholder="Enter your email"
        />
      </div>

      <div>
        <label htmlFor="age">Age:</label>
        <input
          type="number"
          id="age"
          value={age}
          onChange={(e) => setAge(parseInt(e.target.value) || 0)}
          min="0"
          max="120"
        />
```

```
      </div>

      <button type="submit">Submit</button>
    </form>
  );
};
```

Managing Multiple Inputs with Objects

Instead of individual state variables, you can use an object:

```
interface FormData {
  firstName: string;
  lastName: string;
  email: string;
  phone: string;
  country: string;
  agreeToTerms: boolean;
}

const RegistrationForm: React.FC = () => {
  const [formData, setFormData] = useState<FormData>({
    firstName: '',
    lastName: '',
    email: '',
    phone: '',
    country: '',
    agreeToTerms: false
  });

  const handleInputChange = (
    e: React.ChangeEvent<HTMLInputElement | HTMLSelectElement>
  ) => {
    const { name, value, type } = e.target;

    // Handle checkboxes differently
    if (type === 'checkbox') {
      const checked = (e.target as HTMLInputElement).checked;
      setFormData(prev => ({
        ...prev,
        [name]: checked
      }));
    } else {
      setFormData(prev => ({
        ...prev,
        [name]: value
      }));
    }
  };
```

```
const handleSubmit = (e: React.FormEvent<HTMLFormElement>) => {
  e.preventDefault();
  console.log('Registration data:', formData);
};

return (
  <form onSubmit={handleSubmit} className="registration-form">
    <div className="form-row">
      <div>
        <label htmlFor="firstName">First Name:</label>
        <input
          type="text"
          id="firstName"
          name="firstName"
          value={formData.firstName}
          onChange={handleInputChange}
          required
        />
      </div>

      <div>
        <label htmlFor="lastName">Last Name:</label>
        <input
          type="text"
          id="lastName"
          name="lastName"
          value={formData.lastName}
          onChange={handleInputChange}
          required
        />
      </div>
    </div>

    <div>
      <label htmlFor="email">Email:</label>
      <input
        type="email"
        id="email"
        name="email"
        value={formData.email}
        onChange={handleInputChange}
        required
      />
    </div>

    <div>
      <label htmlFor="phone">Phone:</label>
      <input
        type="tel"
```

```
          id="phone"
          name="phone"
          value={formData.phone}
          onChange={handleInputChange}
        />
      </div>

      <div>
        <label htmlFor="country">Country:</label>
        <select
          id="country"
          name="country"
          value={formData.country}
          onChange={handleInputChange}
          required
        >
          <option value="">Select a country</option>
          <option value="us">United States</option>
          <option value="uk">United Kingdom</option>
          <option value="ca">Canada</option>
          <option value="au">Australia</option>
        </select>
      </div>

      <div>
        <label>
          <input
            type="checkbox"
            name="agreeToTerms"
            checked={formData.agreeToTerms}
            onChange={handleInputChange}
            required
          />
          I agree to the terms and conditions
        </label>
      </div>

      <button
        type="submit"
        disabled={!formData.agreeToTerms}
      >
        Register
      </button>
    </form>
  );
};
```

Different Input Types and Their Handlers

```
interface SurveyData {
  name: string;
  rating: number;
  category: string;
  comments: string;
  newsletter: boolean;
  contactMethod: 'email' | 'phone' | 'both';
  tags: string[];
}

const SurveyForm: React.FC = () => {
  const [survey, setSurvey] = useState<SurveyData>({
    name: '',
    rating: 5,
    category: '',
    comments: '',
    newsletter: false,
    contactMethod: 'email',
    tags: []
  });

  const handleInputChange = (
    e: React.ChangeEvent<HTMLInputElement | HTMLSelectElement |
HTMLTextAreaElement>
  ) => {
    const { name, value, type } = e.target;

    if (type === 'checkbox') {
      const checked = (e.target as HTMLInputElement).checked;
      setSurvey(prev => ({ ...prev, [name]: checked }));
    } else if (type === 'number' || type === 'range') {
      setSurvey(prev => ({ ...prev, [name]: parseFloat(value) || 0 }));
    } else {
      setSurvey(prev => ({ ...prev, [name]: value }));
    }
  };

  const handleTagChange = (tag: string) => {
    setSurvey(prev => ({
      ...prev,
      tags: prev.tags.includes(tag)
        ? prev.tags.filter(t => t !== tag)
        : [...prev.tags, tag]
    }));
  };

  const handleSubmit = (e: React.FormEvent<HTMLFormElement>) => {
    e.preventDefault();
```

```
  console.log('Survey data:', survey);
};

const availableTags = ['React', 'TypeScript', 'JavaScript', 'CSS', 'HTML'];

return (
  <form onSubmit={handleSubmit} className="survey-form">
    {/* Text Input */}
    <div>
      <label htmlFor="name">Name:</label>
      <input
        type="text"
        id="name"
        name="name"
        value={survey.name}
        onChange={handleInputChange}
      />
    </div>

    {/* Range Input */}
    <div>
      <label htmlFor="rating">Rating: {survey.rating}/10</label>
      <input
        type="range"
        id="rating"
        name="rating"
        min="1"
        max="10"
        value={survey.rating}
        onChange={handleInputChange}
      />
    </div>

    {/* Select Dropdown */}
    <div>
      <label htmlFor="category">Category:</label>
      <select
        id="category"
        name="category"
        value={survey.category}
        onChange={handleInputChange}
      >
        <option value="">Select category</option>
        <option value="frontend">Frontend</option>
        <option value="backend">Backend</option>
        <option value="fullstack">Full Stack</option>
        <option value="mobile">Mobile</option>
      </select>
    </div>
```

```
{/* Textarea */}
<div>
  <label htmlFor="comments">Comments:</label>
  <textarea
    id="comments"
    name="comments"
    value={survey.comments}
    onChange={handleInputChange}
    rows={4}
    placeholder="Share your thoughts..."
  />
</div>

{/* Checkbox */}
<div>
  <label>
    <input
      type="checkbox"
      name="newsletter"
      checked={survey.newsletter}
      onChange={handleInputChange}
    />
    Subscribe to newsletter
  </label>
</div>

{/* Radio Buttons */}
<div>
  <fieldset>
    <legend>Preferred contact method:</legend>
    {(['email', 'phone', 'both'] as const).map(method => (
      <label key={method}>
        <input
          type="radio"
          name="contactMethod"
          value={method}
          checked={survey.contactMethod === method}
          onChange={handleInputChange}
        />
        {method.charAt(0).toUpperCase() + method.slice(1)}
      </label>
    ))}
  </fieldset>
</div>

{/* Multiple Checkboxes (Tags) */}
<div>
  <fieldset>
```

```
          <legend>Technologies you're interested in:</legend>
          {availableTags.map(tag => (
            <label key={tag}>
              <input
                type="checkbox"
                checked={survey.tags.includes(tag)}
                onChange={() => handleTagChange(tag)}
              />
              {tag}
            </label>
          ))}
        </fieldset>
      </div>

      <button type="submit">Submit Survey</button>
    </form>
  );
};
```

Simple Manual Validation

```
interface ValidationErrors {
  name?: string;
  email?: string;
  password?: string;
  confirmPassword?: string;
}

const SignupForm: React.FC = () => {
  const [formData, setFormData] = useState({
    name: '',
    email: '',
    password: '',
    confirmPassword: ''
  });

  const [errors, setErrors] = useState<ValidationErrors>({});
  const [isSubmitted, setIsSubmitted] = useState(false);

  const validateForm = (): ValidationErrors => {
    const newErrors: ValidationErrors = {};

    // Name validation
    if (!formData.name.trim()) {
      newErrors.name = 'Name is required';
    } else if (formData.name.trim().length < 2) {
      newErrors.name = 'Name must be at least 2 characters';
    }

    // Email validation
```

```
  const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  if (!formData.email) {
    newErrors.email = 'Email is required';
  } else if (!emailRegex.test(formData.email)) {
    newErrors.email = 'Please enter a valid email';
  }

  // Password validation
  if (!formData.password) {
    newErrors.password = 'Password is required';
  } else if (formData.password.length < 8) {
    newErrors.password = 'Password must be at least 8 characters';
  }

  // Confirm password validation
  if (!formData.confirmPassword) {
    newErrors.confirmPassword = 'Please confirm your password';
  } else if (formData.password !== formData.confirmPassword) {
    newErrors.confirmPassword = 'Passwords do not match';
  }

  return newErrors;
};

const handleInputChange = (e: React.ChangeEvent<HTMLInputElement>) => {
  const { name, value } = e.target;
  setFormData(prev => ({ ...prev, [name]: value }));

  // Clear error when user starts typing
  if (errors[name as keyof ValidationErrors]) {
    setErrors(prev => ({ ...prev, [name]: undefined }));
  }
};

const handleSubmit = (e: React.FormEvent<HTMLFormElement>) => {
  e.preventDefault();
  setIsSubmitted(true);

  const validationErrors = validateForm();
  setErrors(validationErrors);

  if (Object.keys(validationErrors).length === 0) {
    console.log('Form is valid!', formData);
    // Handle successful form submission
  }
};

return (
  <form onSubmit={handleSubmit} className="signup-form">
```

```
      <div className="form-group">
        <label htmlFor="name">Name:</label>
        <input
          type="text"
          id="name"
          name="name"
          value={formData.name}
          onChange={handleInputChange}
          className={errors.name ? 'error' : ''}
        />
        {errors.name && <span className="error-message">{errors.name}</span>}
      </div>

      <div className="form-group">
        <label htmlFor="email">Email:</label>
        <input
          type="email"
          id="email"
          name="email"
          value={formData.email}
          onChange={handleInputChange}
          className={errors.email ? 'error' : ''}
        />
        {errors.email && <span className="error-
message">{errors.email}</span>}
      </div>

      <div className="form-group">
        <label htmlFor="password">Password:</label>
        <input
          type="password"
          id="password"
          name="password"
          value={formData.password}
          onChange={handleInputChange}
          className={errors.password ? 'error' : ''}
        />
        {errors.password && <span className="error-
message">{errors.password}</span>}
      </div>

      <div className="form-group">
        <label htmlFor="confirmPassword">Confirm Password:</label>
        <input
          type="password"
          id="confirmPassword"
          name="confirmPassword"
          value={formData.confirmPassword}
          onChange={handleInputChange}
          className={errors.confirmPassword ? 'error' : ''}
```

```
        />
        {errors.confirmPassword && <span className="error-
message">{errors.confirmPassword}</span>}
      </div>

      <button type="submit">Sign Up</button>
    </form>
  );
};
```

💡 Pro Tips

1. **Use controlled components**: They give you more control and are easier to test
2. **Handle all input types**: Different inputs need different change handlers
3. **Validate on both change and submit**: Provide immediate feedback but validate thoroughly on submit
4. **Use generic change handlers**: Reduce code duplication with reusable handlers

🚨 Common Pitfalls

1. **Stale state in handlers**: Using stale values in event handlers
2. **Not preventing default**: Forgetting `e.preventDefault()` in form submission
3. **Incorrect checkbox handling**: Checkboxes use `checked` property, not `value`
4. **Memory leaks with complex state**: Be careful with nested object updates

Exercise: Contact Form with Validation

Create a contact form with the following requirements:

1. Fields: name, email, subject, message, priority (high/medium/low)
2. Validation: all fields required, email format validation
3. Character counter for message (max 500 characters)
4. Success message after submission

```
interface ContactFormData {
  name: string;
  email: string;
  subject: string;
  message: string;
  priority: 'high' | 'medium' | 'low';
}

const ContactForm: React.FC = () => {
  // Your implementation here
  return <div>Contact Form</div>;
};
```

Chapter 12: Forms with React Hook Form

Why Use a Form Library?

While manual form handling works for simple forms, complex forms can become unwieldy. React Hook Form provides:

- **Better performance**: Minimal re-renders
- **Easy validation**: Built-in validation rules
- **TypeScript support**: Excellent type safety
- **Less boilerplate**: Reduces code complexity

Installing React Hook Form

```
npm install react-hook-form
```

Basic React Hook Form Usage

```
import { useForm, SubmitHandler } from 'react-hook-form';

interface LoginFormData {
  email: string;
  password: string;
}

const LoginForm: React.FC = () => {
  const {
    register,
    handleSubmit,
    formState: { errors, isSubmitting }
  } = useForm<LoginFormData>();

  const onSubmit: SubmitHandler<LoginFormData> = async (data) => {
    console.log('Form data:', data);

    // Simulate API call
    await new Promise(resolve => setTimeout(resolve, 2000));

    alert('Login successful!');
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <div>
        <label htmlFor="email">Email:</label>
        <input
          type="email"
          id="email"
          {...register('email', {
            required: 'Email is required',
```

```
            pattern: {
              value: /^[^\s@]+@[^\s@]+\.[^\s@]+$/,
              message: 'Please enter a valid email'
            }
          })}
        />
        {errors.email && <span
className="error">{errors.email.message}</span>}
      </div>

      <div>
        <label htmlFor="password">Password:</label>
        <input
          type="password"
          id="password"
          {...register('password', {
            required: 'Password is required',
            minLength: {
              value: 8,
              message: 'Password must be at least 8 characters'
            }
          })}
        />
        {errors.password && <span
className="error">{errors.password.message}</span>}
      </div>

      <button type="submit" disabled={isSubmitting}>
        {isSubmitting ? 'Logging in...' : 'Login'}
      </button>
    </form>
  );
};
```

Advanced Form with Complex Validation

```
import { useForm, SubmitHandler, Controller } from 'react-hook-form';

interface UserProfileData {
  firstName: string;
  lastName: string;
  email: string;
  age: number;
  country: string;
  bio: string;
  newsletter: boolean;
  skills: string[];
}

const UserProfileForm: React.FC = () => {
```

```
  const {
    register,
    handleSubmit,
    control,
    watch,
    setValue,
    formState: { errors, isSubmitting, isDirty, isValid }
  } = useForm<UserProfileData>({
    defaultValues: {
      firstName: '',
      lastName: '',
      email: '',
      age: 18,
      country: '',
      bio: '',
      newsletter: false,
      skills: []
    },
    mode: 'onChange' // Validate on change
  });

  const watchedSkills = watch('skills');
  const availableSkills = ['React', 'TypeScript', 'Node.js', 'Python',
'CSS'];

  const onSubmit: SubmitHandler<UserProfileData> = async (data) => {
    console.log('Profile data:', data);

    try {
      // Simulate API call
      await new Promise((resolve, reject) => {
        setTimeout(() => {
          if (Math.random() > 0.1) { // 90% success rate
            resolve('Success');
          } else {
            reject(new Error('Network error'));
          }
        }, 2000);
      });

      alert('Profile updated successfully!');
    } catch (error) {
      alert('Failed to update profile. Please try again.');
    }
  };

  const handleSkillToggle = (skill: string) => {
    const currentSkills = watchedSkills || [];
    const newSkills = currentSkills.includes(skill)
```

```
      ? currentSkills.filter(s => s !== skill)
      : [...currentSkills, skill];

    setValue('skills', newSkills, { shouldValidate: true });
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)} className="profile-form">
      <div className="form-row">
        <div>
          <label htmlFor="firstName">First Name:</label>
          <input
            type="text"
            id="firstName"
            {...register('firstName', {
              required: 'First name is required',
              minLength: {
                value: 2,
                message: 'First name must be at least 2 characters'
              },
              pattern: {
                value: /^[A-Za-z]+$/,
                message: 'First name can only contain letters'
              }
            })}
          />
          {errors.firstName && <span
className="error">{errors.firstName.message}</span>}
        </div>

        <div>
          <label htmlFor="lastName">Last Name:</label>
          <input
            type="text"
            id="lastName"
            {...register('lastName', {
              required: 'Last name is required',
              minLength: {
                value: 2,
                message: 'Last name must be at least 2 characters'
              }
            })}
          />
          {errors.lastName && <span
className="error">{errors.lastName.message}</span>}
        </div>
      </div>

      <div>
```

```
        <label htmlFor="email">Email:</label>
        <input
          type="email"
          id="email"
          {...register('email', {
            required: 'Email is required',
            pattern: {
              value: /^[^\s@]+@[^\s@]+\.[^\s@]+$/,
              message: 'Please enter a valid email'
            }
          })}
        />
        {errors.email && <span
className="error">{errors.email.message}</span>}
      </div>

      <div>
        <label htmlFor="age">Age:</label>
        <input
          type="number"
          id="age"
          {...register('age', {
            required: 'Age is required',
            min: {
              value: 13,
              message: 'You must be at least 13 years old'
            },
            max: {
              value: 120,
              message: 'Age must be realistic'
            }
          })}
        />
        {errors.age && <span className="error">{errors.age.message}</span>}
      </div>

      <div>
        <label htmlFor="country">Country:</label>
        <select
          id="country"
          {...register('country', {
            required: 'Please select a country'
          })}
        >
          <option value="">Select a country</option>
          <option value="us">United States</option>
          <option value="uk">United Kingdom</option>
          <option value="ca">Canada</option>
          <option value="au">Australia</option>
          <option value="de">Germany</option>
```

```
        </select>
        {errors.country && <span
className="error">{errors.country.message}</span>}
      </div>

      <div>
        <label htmlFor="bio">Bio:</label>
        <Controller
          name="bio"
          control={control}
          rules={{
            maxLength: {
              value: 500,
              message: 'Bio cannot exceed 500 characters'
            }
          }}
          render={({ field }) => (
            <div>
              <textarea
                {...field}
                id="bio"
                rows={4}
                placeholder="Tell us about yourself..."
              />
              <small>{field.value.length}/500 characters</small>
            </div>
          )}
        />
        {errors.bio && <span className="error">{errors.bio.message}</span>}
      </div>

      <div>
        <label>Skills:</label>
        <div className="skills-grid">
          {availableSkills.map(skill => (
            <label key={skill} className="skill-item">
              <input
                type="checkbox"
                checked={watchedSkills?.includes(skill) || false}
                onChange={() => handleSkillToggle(skill)}
              />
              {skill}
            </label>
          ))}
        </div>
      </div>

      <div>
        <label>
          <input
```

```
            type="checkbox"
            {...register('newsletter')}
          />
          Subscribe to newsletter
        </label>
      </div>

      <div className="form-actions">
        <button
          type="submit"
          disabled={isSubmitting || !isDirty || !isValid}
        >
          {isSubmitting ? 'Updating...' : 'Update Profile'}
        </button>

        {isDirty && (
          <p className="info">You have unsaved changes</p>
        )}
      </div>
    </form>
  );
};
```

Type-Safe Validation with Zod

For even better validation, you can use Zod with React Hook Form:

```
npm install @hookform/resolvers zod
```

```
import { useForm } from 'react-hook-form';
import { zodResolver } from '@hookform/resolvers/zod';
import { z } from 'zod';

// Define validation schema
const signupSchema = z.object({
  username: z
    .string()
    .min(3, 'Username must be at least 3 characters')
    .max(20, 'Username cannot exceed 20 characters')
    .regex(/^[a-zA-Z0-9_]+$/, 'Username can only contain letters, numbers,
and underscores'),

  email: z
    .string()
    .email('Please enter a valid email address'),

  password: z
    .string()
    .min(8, 'Password must be at least 8 characters')
    .regex(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/, 'Password must contain at least
```

```
one uppercase letter, one lowercase letter, and one number'),

  confirmPassword: z.string(),

  age: z
    .number()
    .min(13, 'You must be at least 13 years old')
    .max(120, 'Please enter a valid age'),

  terms: z
    .boolean()
    .refine(val => val === true, 'You must accept the terms and conditions')
}).refine(data => data.password === data.confirmPassword, {
  message: "Passwords don't match",
  path: ["confirmPassword"]
});

type SignupFormData = z.infer<typeof signupSchema>;

const SignupForm: React.FC = () => {
  const {
    register,
    handleSubmit,
    formState: { errors, isSubmitting }
  } = useForm<SignupFormData>({
    resolver: zodResolver(signupSchema),
    mode: 'onChange'
  });

  const onSubmit = async (data: SignupFormData) => {
    console.log('Signup data:', data);
    // Handle form submission
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <div>
        <label htmlFor="username">Username:</label>
        <input
          type="text"
          id="username"
          {...register('username')}
        />
        {errors.username && <span
className="error">{errors.username.message}</span>}
      </div>

      <div>
        <label htmlFor="email">Email:</label>
```

```
        <input
          type="email"
          id="email"
          {...register('email')}
        />
        {errors.email && <span
className="error">{errors.email.message}</span>}
      </div>

      <div>
        <label htmlFor="password">Password:</label>
        <input
          type="password"
          id="password"
          {...register('password')}
        />
        {errors.password && <span
className="error">{errors.password.message}</span>}
      </div>

      <div>
        <label htmlFor="confirmPassword">Confirm Password:</label>
        <input
          type="password"
          id="confirmPassword"
          {...register('confirmPassword')}
        />
        {errors.confirmPassword && <span
className="error">{errors.confirmPassword.message}</span>}
      </div>

      <div>
        <label htmlFor="age">Age:</label>
        <input
          type="number"
          id="age"
          {...register('age', { valueAsNumber: true })}
        />
        {errors.age && <span className="error">{errors.age.message}</span>}
      </div>

      <div>
        <label>
          <input
            type="checkbox"
            {...register('terms')}
          />
          I accept the terms and conditions
        </label>
```

```
        {errors.terms && <span
className="error">{errors.terms.message}</span>}
      </div>

      <button type="submit" disabled={isSubmitting}>
        {isSubmitting ? 'Creating Account...' : 'Sign Up'}
      </button>
    </form>
  );
};
```

Custom Input Components

Create reusable input components with React Hook Form:

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

interface InputProps extends React.InputHTMLAttributes<HTMLInputElement> {
  label: string;
  error?: string;
}

const Input = forwardRef<HTMLInputElement, InputProps>(
  ({ label, error, ...props }, ref) => {
    return (
      <div className="input-group">
        <label htmlFor={props.id}>{label}</label>
        <input
          ref={ref}
          {...props}
          className={`input ${error ? 'input-error' : ''}`}
        />
        {error && <span className="error-message">{error}</span>}
      </div>
    );
  }
);

Input.displayName = 'Input';

// Usage with React Hook Form
const CustomForm: React.FC = () => {
  const {
    register,
    handleSubmit,
    formState: { errors }
  } = useForm<{
    name: string;
    email: string;
  }>();
```

```
  return (
    <form onSubmit={handleSubmit(data => console.log(data))}>
      <Input
        label="Name"
        id="name"
        {...register('name', { required: 'Name is required' })}
        error={errors.name?.message}
      />

      <Input
        label="Email"
        id="email"
        type="email"
        {...register('email', {
          required: 'Email is required',
          pattern: {
            value: /^[^\s@]+@[^\s@]+\.[^\s@]+$/,
            message: 'Invalid email format'
          }
        })}
        error={errors.email?.message}
      />

      <button type="submit">Submit</button>
    </form>
  );
};
```

Pro Tips

1. **Use mode: 'onChange'**: For better user experience with real-time validation
2. **Leverage TypeScript**: Use proper types for better development experience
3. **Create reusable components**: Build a library of form components
4. **Handle loading states**: Show feedback during form submission

Common Pitfalls

1. **Not using Controller for complex inputs**: Some inputs need the Controller component
2. **Forgetting to handle submission errors**: Always provide error feedback
3. **Over-validating**: Don't make forms too restrictive
4. **Not using proper TypeScript types**: Missing out on type safety benefits

Exercise: Job Application Form

Create a comprehensive job application form with:

1. Personal info (name, email, phone)

2. File upload for resume (use Controller)
3. Work experience (dynamic list - add/remove entries)
4. Skills selection (checkboxes)
5. Cover letter (textarea with character limit)
6. Full validation with Zod
7. Form submission with loading state

```
interface JobApplicationData {
  // Define your interface here
}

const JobApplicationForm: React.FC = () => {
  // Your implementation here
  return <div>Job Application Form</div>;
};
```

Chapter 13: Routing with React Router v6+

What is Routing?

In traditional websites, when you click a link or type a URL, the browser makes a request to the server for a new page. In React applications, we use **client-side routing** - the browser doesn't reload the page, but JavaScript changes what's displayed based on the URL.

Think of routing like having multiple rooms in a house. Each room (route) shows different content, but you're still in the same house (React app).

Why Do We Need Routing?

Without routing, your React app would be a Single Page Application (SPA) that shows the same content regardless of the URL. Routing allows you to:

- Create different "pages" in your app
- Share specific URLs with others
- Use browser back/forward buttons
- Organize your app into logical sections

Installing React Router

First, let's install React Router:

```
npm install react-router-dom
npm install -D @types/react-router-dom  # TypeScript types
```

Basic Router Setup

Let's create a simple multi-page app with Home, About, and Contact pages.

src/App.tsx

```
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';
import './App.css';

// Page components
const Home = () => (
  <div>
    <h1>Welcome to Our App</h1>
    <p>This is the home page!</p>
  </div>
);

const About = () => (
  <div>
    <h1>About Us</h1>
    <p>Learn more about our company here.</p>
  </div>
);

const Contact = () => (
  <div>
    <h1>Contact Us</h1>
    <p>Get in touch with us!</p>
    <form>
      <input type="email" placeholder="Your email" />
      <textarea placeholder="Your message"></textarea>
      <button type="submit">Send</button>
    </form>
  </div>
);

// Navigation component
const Navigation = () => (
  <nav style={{ padding: '1rem', borderBottom: '1px solid #ccc' }}>
    <Link to="/" style={{ marginRight: '1rem' }}>Home</Link>
    <Link to="/about" style={{ marginRight: '1rem' }}>About</Link>
    <Link to="/contact">Contact</Link>
  </nav>
);

function App() {
  return (
    <BrowserRouter>
      <div className="App">
        <Navigation />
        <main style={{ padding: '2rem' }}>
          <Routes>
            <Route path="/" element={<Home />} />
            <Route path="/about" element={<About />} />
```

```
            <Route path="/contact" element={<Contact />} />
          </Routes>
        </main>
      </div>
    </BrowserRouter>
  );
}

export default App;
```

Understanding the Router Components

BrowserRouter

- Wraps your entire app
- Manages the browser's history and URL changes
- Must be the parent of all routing components

Routes and Route

- `Routes` is a container for all your route definitions
- Each `Route` defines a path and what component to render
- The `path` prop matches against the URL
- The `element` prop is the component to render

Link

- Creates navigation links that don't reload the page
- Similar to `<a>` tags but for client-side routing
- The `to` prop specifies the destination path

Route Parameters

Sometimes you need dynamic routes that can handle different values. For example, viewing a specific user profile or blog post.

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

// User profile component
const UserProfile = () => {
  const { userId } = useParams<{ userId: string }>();

  return (
    <div>
      <h1>User Profile</h1>
      <p>Viewing profile for user ID: {userId}</p>
    </div>
  );
};

// Blog post component
```

```
const BlogPost = () => {
  const { postId } = useParams<{ postId: string }>();

  return (
    <div>
      <h1>Blog Post</h1>
      <p>Reading post ID: {postId}</p>
    </div>
  );
};

// Add these routes to your Routes component
<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/about" element={<About />} />
  <Route path="/contact" element={<Contact />} />
  <Route path="/user/:userId" element={<UserProfile />} />
  <Route path="/blog/:postId" element={<BlogPost />} />
</Routes>
```

The `:userId` and `:postId` are **route parameters**. When someone visits `/user/123`, the `userId` parameter will be `"123"`.

Navigation with useNavigate

Sometimes you need to navigate programmatically (like after form submission):

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

const ContactForm = () => {
  const navigate = useNavigate();

  const handleSubmit = (e: React.FormEvent) => {
    e.preventDefault();
    // Process form submission...

    // Navigate to success page
    navigate('/contact/success');
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="email" placeholder="Your email" required />
      <textarea placeholder="Your message" required></textarea>
      <button type="submit">Send Message</button>
    </form>
  );
};
```

Nested Routes

You can nest routes for more complex layouts:

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

const BlogLayout = () => (
  <div>
    <h1>Our Blog</h1>
    <nav>
      <Link to="/blog">All Posts</Link>
      <Link to="/blog/create">Create Post</Link>
    </nav>
    <div style={{ marginTop: '1rem' }}>
      <Outlet /> {/* Child routes render here */}
    </div>
  </div>
);

const BlogList = () => (
  <div>
    <h2>All Blog Posts</h2>
    <p>List of blog posts would go here...</p>
  </div>
);

const CreatePost = () => (
  <div>
    <h2>Create New Post</h2>
    <form>
      <input type="text" placeholder="Post title" />
      <textarea placeholder="Post content"></textarea>
      <button type="submit">Create Post</button>
    </form>
  </div>
);

// In your Routes:
<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/blog" element={<BlogLayout />}>
    <Route index element={<BlogList />} />
    <Route path="create" element={<CreatePost />} />
  </Route>
</Routes>
```

404 - Not Found Pages

Handle routes that don't match any defined paths:

```
const NotFound = () => (
  <div style={{ textAlign: 'center', padding: '2rem' }}>
    <h1>404 - Page Not Found</h1>
    <p>The page you're looking for doesn't exist.</p>
    <Link to="/">Go back to Home</Link>
  </div>
);

// Add this as the last route (it matches everything)
<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/about" element={<About />} />
  <Route path="/contact" element={<Contact />} />
  <Route path="*" element={<NotFound />} />
</Routes>
```

💡 Pro Tips

1. **Always wrap your app in `BrowserRouter`** - This should be at the root level
2. **Use `Link` instead of `<a>` tags** - Links prevent page reloads
3. **Type your route parameters** - Use generics with `useParams<{ paramName: string }>()`
4. **Plan your route structure** - Think about your app's navigation flow before coding

⚠️ Common Pitfalls

1. **Forgetting BrowserRouter**: Routes won't work without this wrapper
2. **Using `<a>` tags**: These will reload the page instead of using client-side routing
3. **Not handling 404s**: Always include a catch-all route for unmatched paths
4. **Complex nested routing too early**: Start simple, add complexity as needed

🏋️ Chapter 13 Exercise

Create a simple portfolio website with routing:

1. Create a `Portfolio` component with navigation to:
   - Home (`/`)
   - About (`/about`)
   - Projects (`/projects`)
   - Contact (`/contact`)
2. Add a dynamic route for individual projects (`/projects/:projectId`)
3. Include a 404 page for unmatched routes
4. Use `useNavigate` to redirect users after they submit the contact form

Bonus: Add nested routes under `/projects` for different categories (web, mobile, etc.)

Chapter 14: Mini Project - Task Tracker App

Welcome to the final chapter! 🎉 It's time to put everything you've learned into practice by building a complete **Task Tracker Application**. This project will incorporate all the concepts we've covered: components, props, state, effects, routing, data fetching, and forms with React Hook Form.

📋 What We're Building

Our Task Tracker will have the following features:

- **Dashboard**: Overview of tasks with statistics
- **Task List**: View all tasks with filtering and sorting
- **Add Task Form**: Create new tasks with validation
- **Edit Task**: Update existing tasks
- **Task Details**: View individual task information
- **Categories**: Organize tasks by categories
- **Local Storage**: Persist data between sessions

🏗️ Project Structure

```
task-tracker/
├── src/
│   ├── components/
│   │   ├── common/
│   │   │   ├── Header.tsx
│   │   │   ├── Navigation.tsx
│   │   │   └── LoadingSpinner.tsx
│   │   ├── tasks/
│   │   │   ├── TaskCard.tsx
│   │   │   ├── TaskList.tsx
│   │   │   ├── TaskForm.tsx
│   │   │   └── TaskFilters.tsx
│   │   └── dashboard/
│   │       ├── StatsCard.tsx
│   │       └── RecentTasks.tsx
│   ├── pages/
│   │   ├── Dashboard.tsx
│   │   ├── Tasks.tsx
│   │   ├── AddTask.tsx
│   │   ├── EditTask.tsx
│   │   └── TaskDetails.tsx
│   ├── hooks/
│   │   ├── useTasks.ts
│   │   └── useLocalStorage.ts
```

```
│   ├── types/
│   │   └── index.ts
│   ├── utils/
│   │   └── taskUtils.ts
│   └── App.tsx
```

🔧 Setting Up the Project

First, let's create our project and install dependencies:

```
npm create vite@latest task-tracker -- --template react-ts
cd task-tracker
npm install
npm install react-router-dom react-hook-form @types/react-router-dom
npm install lucide-react # For icons
```

📝 Defining Types

Let's start by defining our TypeScript types in `src/types/index.ts`:

```
// src/types/index.ts
export type TaskPriority = 'low' | 'medium' | 'high';
export type TaskStatus = 'pending' | 'in-progress' | 'completed';
export type TaskCategory = 'work' | 'personal' | 'health' | 'learning' |
'other';

export interface Task {
  id: string;
  title: string;
  description: string;
  priority: TaskPriority;
  status: TaskStatus;
  category: TaskCategory;
  dueDate?: string;
  createdAt: string;
  updatedAt: string;
}

export interface TaskFormData {
  title: string;
  description: string;
  priority: TaskPriority;
  category: TaskCategory;
  dueDate?: string;
}

export interface TaskFilters {
  status?: TaskStatus;
  priority?: TaskPriority;
  category?: TaskCategory;
```

```
  search?: string;
}

export interface TaskStats {
  total: number;
  completed: number;
  pending: number;
  inProgress: number;
  overdue: number;
}
```

🪝 Custom Hooks

useLocalStorage Hook

```
// src/hooks/useLocalStorage.ts
import { useState, useEffect } from 'react';

export function useLocalStorage<T>(
  key: string,
  initialValue: T
): [T, (value: T | ((val: T) => T)) => void] {
  // Get value from localStorage or use initial value
  const [storedValue, setStoredValue] = useState<T>(() => {
    try {
      const item = window.localStorage.getItem(key);
      return item ? JSON.parse(item) : initialValue;
    } catch (error) {
      console.error(`Error reading localStorage key "${key}":`, error);
      return initialValue;
    }
  });

  // Update localStorage when state changes
  const setValue = (value: T | ((val: T) => T)) => {
    try {
      // Allow value to be a function for useState-like API
      const valueToStore = value instanceof Function ? value(storedValue) :
value;
      setStoredValue(valueToStore);
      window.localStorage.setItem(key, JSON.stringify(valueToStore));
    } catch (error) {
      console.error(`Error setting localStorage key "${key}":`, error);
    }
  };

  return [storedValue, setValue];
}
```

useTasks Hook

```
// src/hooks/useTasks.ts
import { useMemo } from 'react';
import { Task, TaskFilters, TaskStats } from '../types';
import { useLocalStorage } from './useLocalStorage';
import { generateId, isOverdue } from '../utils/taskUtils';

export function useTasks() {
  const [tasks, setTasks] = useLocalStorage<Task[]>('tasks', []);

  const addTask = (taskData: Omit<Task, 'id' | 'createdAt' | 'updatedAt'>) =>
{
    const newTask: Task = {
      ...taskData,
      id: generateId(),
      status: 'pending',
      createdAt: new Date().toISOString(),
      updatedAt: new Date().toISOString(),
    };
    setTasks(prev => [...prev, newTask]);
    return newTask;
  };

  const updateTask = (id: string, updates: Partial<Task>) => {
    setTasks(prev =>
      prev.map(task =>
        task.id === id
          ? { ...task, ...updates, updatedAt: new Date().toISOString() }
          : task
      )
    );
  };

  const deleteTask = (id: string) => {
    setTasks(prev => prev.filter(task => task.id !== id));
  };

  const getTaskById = (id: string): Task | undefined => {
    return tasks.find(task => task.id === id);
  };

  const getFilteredTasks = (filters: TaskFilters): Task[] => {
    return tasks.filter(task => {
      if (filters.status && task.status !== filters.status) return false;
      if (filters.priority && task.priority !== filters.priority) return
false;
      if (filters.category && task.category !== filters.category) return
false;
      if (filters.search) {
```

```
        const searchLower = filters.search.toLowerCase();
        return (
          task.title.toLowerCase().includes(searchLower) ||
          task.description.toLowerCase().includes(searchLower)
        );
      }
      return true;
    });
  };

  const getTaskStats = (): TaskStats => {
    const total = tasks.length;
    const completed = tasks.filter(task => task.status ===
'completed').length;
    const pending = tasks.filter(task => task.status === 'pending').length;
    const inProgress = tasks.filter(task => task.status === 'in-
progress').length;
    const overdue = tasks.filter(task =>
      task.status !== 'completed' && task.dueDate && isOverdue(task.dueDate)
    ).length;

    return { total, completed, pending, inProgress, overdue };
  };

  const stats = useMemo(() => getTaskStats(), [tasks]);

  return {
    tasks,
    addTask,
    updateTask,
    deleteTask,
    getTaskById,
    getFilteredTasks,
    stats,
  };
}
```

🛠 Utility Functions

```
// src/utils/taskUtils.ts
export function generateId(): string {
  return Date.now().toString(36) + Math.random().toString(36).substring(2);
}

export function isOverdue(dueDate: string): boolean {
  return new Date(dueDate) < new Date();
}

export function formatDate(dateString: string): string {
  return new Date(dateString).toLocaleDateString('en-US', {
```

```
    year: 'numeric',
    month: 'short',
    day: 'numeric',
  });
}

export function getPriorityColor(priority: string): string {
  switch (priority) {
    case 'high': return 'text-red-600 bg-red-100';
    case 'medium': return 'text-yellow-600 bg-yellow-100';
    case 'low': return 'text-green-600 bg-green-100';
    default: return 'text-gray-600 bg-gray-100';
  }
}

export function getStatusColor(status: string): string {
  switch (status) {
    case 'completed': return 'text-green-600 bg-green-100';
    case 'in-progress': return 'text-blue-600 bg-blue-100';
    case 'pending': return 'text-gray-600 bg-gray-100';
    default: return 'text-gray-600 bg-gray-100';
  }
}
```

🧩 Core Components

Header Component

```
// src/components/common/Header.tsx
import React from 'react';
import { Link } from 'react-router-dom';
import { CheckSquare } from 'lucide-react';

export const Header: React.FC = () => {
  return (
    <header className="bg-blue-600 text-white shadow-lg">
      <div className="container mx-auto px-4 py-4">
        <div className="flex items-center justify-between">
          <Link to="/" className="flex items-center space-x-2">
            <CheckSquare size={28} />
            <h1 className="text-2xl font-bold">Task Tracker</h1>
          </Link>
          <nav className="flex space-x-6">
            <Link
              to="/"
              className="hover:text-blue-200 transition-colors duration-200"
            >
              Dashboard
            </Link>
            <Link
```

```
              to="/tasks"
              className="hover:text-blue-200 transition-colors duration-200"
            >
              Tasks
            </Link>
            <Link
              to="/add-task"
              className="bg-blue-700 px-4 py-2 rounded-lg hover:bg-blue-800
transition-colors duration-200"
            >
              Add Task
            </Link>
          </nav>
        </div>
      </div>
    </header>
  );
};
```

Task Card Component

```
// src/components/tasks/TaskCard.tsx
import React from 'react';
import { Link } from 'react-router-dom';
import { Calendar, Edit, Trash2, Clock } from 'lucide-react';
import { Task } from '../../types';
import { formatDate, getPriorityColor, getStatusColor, isOverdue } from
'../../utils/taskUtils';

interface TaskCardProps {
  task: Task;
  onDelete: (id: string) => void;
  onStatusChange: (id: string, status: Task['status']) => void;
}

export const TaskCard: React.FC<TaskCardProps> = ({ task, onDelete,
onStatusChange }) => {
  const handleStatusChange = (e: React.ChangeEvent<HTMLSelectElement>) => {
    onStatusChange(task.id, e.target.value as Task['status']);
  };

  const handleDelete = () => {
    if (window.confirm('Are you sure you want to delete this task?')) {
      onDelete(task.id);
    }
  };

  return (
```

```
    <div className="bg-white rounded-lg shadow-md p-6 hover:shadow-lg
transition-shadow duration-200">
      <div className="flex justify-between items-start mb-4">
        <Link
          to={`/tasks/${task.id}`}
          className="text-xl font-semibold text-gray-800 hover:text-blue-600
transition-colors"
        >
          {task.title}
        </Link>
        <div className="flex space-x-2">
          <Link
            to={`/edit-task/${task.id}`}
            className="text-gray-500 hover:text-blue-600 transition-colors"
          >
            <Edit size={18} />
          </Link>
          <button
            onClick={handleDelete}
            className="text-gray-500 hover:text-red-600 transition-colors"
          >
            <Trash2 size={18} />
          </button>
        </div>
      </div>

      <p className="text-gray-600 mb-4 line-clamp-2">{task.description}</p>

      <div className="flex flex-wrap gap-2 mb-4">
        <span className={`px-2 py-1 rounded-full text-xs font-medium
${getPriorityColor(task.priority)}`}>
          {task.priority.charAt(0).toUpperCase() + task.priority.slice(1)}
        </span>
        <span className={`px-2 py-1 rounded-full text-xs font-medium
${getStatusColor(task.status)}`}>
          {task.status.replace('-', ' ').replace(/\b\w/g, l =>
l.toUpperCase())}
        </span>
        <span className="px-2 py-1 rounded-full text-xs font-medium text-
purple-600 bg-purple-100">
          {task.category.charAt(0).toUpperCase() + task.category.slice(1)}
        </span>
      </div>

      {task.dueDate && (
        <div className={`flex items-center space-x-1 mb-4 text-sm ${
          isOverdue(task.dueDate) && task.status !== 'completed'
            ? 'text-red-600'
            : 'text-gray-500'
        }`}>
```

```
          {isOverdue(task.dueDate) && task.status !== 'completed' ? (
            <Clock size={16} />
          ) : (
            <Calendar size={16} />
          )}
          <span>
            {isOverdue(task.dueDate) && task.status !== 'completed' ?
'Overdue: ' : 'Due: '}
            {formatDate(task.dueDate)}
          </span>
        </div>
      )}

      <div className="flex justify-between items-center">
        <select
          value={task.status}
          onChange={handleStatusChange}
          className="px-3 py-1 rounded border border-gray-300 text-sm
focus:outline-none focus:ring-2 focus:ring-blue-500"
        >
          <option value="pending">Pending</option>
          <option value="in-progress">In Progress</option>
          <option value="completed">Completed</option>
        </select>
        <span className="text-xs text-gray-400">
          Updated {formatDate(task.updatedAt)}
        </span>
      </div>
    </div>
  );
};
```

Task Form Component (with React Hook Form)

```
// src/components/tasks/TaskForm.tsx
import React from 'react';
import { useForm } from 'react-hook-form';
import { useNavigate } from 'react-router-dom';
import { TaskFormData, Task, TaskPriority, TaskCategory } from '../../types';

interface TaskFormProps {
  initialData?: Partial<Task>;
  onSubmit: (data: TaskFormData) => void;
  submitButtonText?: string;
}

export const TaskForm: React.FC<TaskFormProps> = ({
  initialData,
  onSubmit,
  submitButtonText = 'Create Task'
```

```
}) => {
  const navigate = useNavigate();

  const {
    register,
    handleSubmit,
    formState: { errors, isSubmitting },
    reset
  } = useForm<TaskFormData>({
    defaultValues: {
      title: initialData?.title || '',
      description: initialData?.description || '',
      priority: initialData?.priority || 'medium',
      category: initialData?.category || 'personal',
      dueDate: initialData?.dueDate || '',
    }
  });

  const handleFormSubmit = async (data: TaskFormData) => {
    try {
      await onSubmit(data);
      if (!initialData) {
        reset(); // Only reset for new tasks
      }
      navigate('/tasks');
    } catch (error) {
      console.error('Error submitting form:', error);
    }
  };

  return (
    <div className="max-w-2xl mx-auto bg-white rounded-lg shadow-md p-8">
      <form onSubmit={handleSubmit(handleFormSubmit)} className="space-y-6">
        {/* Title Field */}
        <div>
          <label htmlFor="title" className="block text-sm font-medium text-
gray-700 mb-2">
            Task Title *
          </label>
          <input
            type="text"
            id="title"
            {...register('title', {
              required: 'Task title is required',
              minLength: {
                value: 3,
                message: 'Title must be at least 3 characters long'
              },
              maxLength: {
                value: 100,
```

```
                message: 'Title must be less than 100 characters'
              }
            })}
            className={`w-full px-4 py-2 border rounded-lg focus:outline-none
focus:ring-2 focus:ring-blue-500 ${
              errors.title ? 'border-red-500' : 'border-gray-300'
            }`}
            placeholder="Enter task title..."
          />
          {errors.title && (
            <p className="mt-1 text-sm text-red-
600">{errors.title.message}</p>
          )}
        </div>

        {/* Description Field */}
        <div>
          <label htmlFor="description" className="block text-sm font-medium
text-gray-700 mb-2">
            Description *
          </label>
          <textarea
            id="description"
            rows={4}
            {...register('description', {
              required: 'Task description is required',
              minLength: {
                value: 10,
                message: 'Description must be at least 10 characters long'
              }
            })}
            className={`w-full px-4 py-2 border rounded-lg focus:outline-none
focus:ring-2 focus:ring-blue-500 ${
              errors.description ? 'border-red-500' : 'border-gray-300'
            }`}
            placeholder="Describe your task..."
          />
          {errors.description && (
            <p className="mt-1 text-sm text-red-
600">{errors.description.message}</p>
          )}
        </div>

        {/* Priority and Category Row */}
        <div className="grid grid-cols-1 md:grid-cols-2 gap-6">
          <div>
            <label htmlFor="priority" className="block text-sm font-medium
text-gray-700 mb-2">
              Priority
            </label>
```

```
            <select
              id="priority"
              {...register('priority')}
              className="w-full px-4 py-2 border border-gray-300 rounded-lg
focus:outline-none focus:ring-2 focus:ring-blue-500"
            >
              <option value="low">Low</option>
              <option value="medium">Medium</option>
              <option value="high">High</option>
            </select>
          </div>

          <div>
            <label htmlFor="category" className="block text-sm font-medium
text-gray-700 mb-2">
              Category
            </label>
            <select
              id="category"
              {...register('category')}
              className="w-full px-4 py-2 border border-gray-300 rounded-lg
focus:outline-none focus:ring-2 focus:ring-blue-500"
            >
              <option value="work">Work</option>
              <option value="personal">Personal</option>
              <option value="health">Health</option>
              <option value="learning">Learning</option>
              <option value="other">Other</option>
            </select>
          </div>
        </div>

        {/* Due Date Field */}
        <div>
          <label htmlFor="dueDate" className="block text-sm font-medium text-
gray-700 mb-2">
            Due Date (Optional)
          </label>
          <input
            type="date"
            id="dueDate"
            {...register('dueDate', {
              validate: (value) => {
                if (!value) return true; // Optional field
                const selectedDate = new Date(value);
                const today = new Date();
                today.setHours(0, 0, 0, 0);

                if (selectedDate < today) {
                  return 'Due date cannot be in the past';
```

```
                }
                return true;
              }
            })}
            className={`w-full px-4 py-2 border rounded-lg focus:outline-none
focus:ring-2 focus:ring-blue-500 ${
              errors.dueDate ? 'border-red-500' : 'border-gray-300'
            }`}
          />
          {errors.dueDate && (
            <p className="mt-1 text-sm text-red-
600">{errors.dueDate.message}</p>
          )}
        </div>

        {/* Form Actions */}
        <div className="flex space-x-4 pt-6">
          <button
            type="submit"
            disabled={isSubmitting}
            className="flex-1 bg-blue-600 text-white py-2 px-4 rounded-lg
hover:bg-blue-700 focus:outline-none focus:ring-2 focus:ring-blue-500
disabled:opacity-50 disabled:cursor-not-allowed transition-colors duration-
200"
          >
            {isSubmitting ? 'Submitting...' : submitButtonText}
          </button>
          <button
            type="button"
            onClick={() => navigate('/tasks')}
            className="flex-1 bg-gray-300 text-gray-700 py-2 px-4 rounded-lg
hover:bg-gray-400 focus:outline-none focus:ring-2 focus:ring-gray-500
transition-colors duration-200"
          >
            Cancel
          </button>
        </div>
      </form>
    </div>
  );
};
```

Pages

Dashboard Page

```
// src/pages/Dashboard.tsx
import React from 'react';
import { Link } from 'react-router-dom';
import { useTasks } from '../hooks/useTasks';
```

```
import { TaskCard } from '../components/tasks/TaskCard';
import { Plus, CheckCircle, Clock, AlertCircle, TrendingUp } from 'lucide-
react';

export const Dashboard: React.FC = () => {
  const { tasks, updateTask, deleteTask, stats } = useTasks();

  // Get recent tasks (last 5 created)
  const recentTasks = tasks
    .sort((a, b) => new Date(b.createdAt).getTime() - new
Date(a.createdAt).getTime())
    .slice(0, 5);

  const handleStatusChange = (id: string, status: 'pending' | 'in-progress' |
'completed') => {
    updateTask(id, { status });
  };

  return (
    <div className="container mx-auto px-4 py-8">
      <div className="mb-8">
        <h1 className="text-3xl font-bold text-gray-800 mb-2">All Tasks</h1>
        <p className="text-gray-600">Manage and organize your tasks
efficiently.</p>
      </div>

      {/* Filters Section */}
      <div className="bg-white rounded-lg shadow-md p-6 mb-8">
        <div className="flex items-center mb-4">
          <Filter size={20} className="mr-2 text-gray-600" />
          <h2 className="text-lg font-semibold text-gray-800">Filters</h2>
        </div>

        <div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-4 gap-4
mb-4">
          {/* Search */}
          <div className="relative">
            <Search size={18} className="absolute left-3 top-1/2 transform -
translate-y-1/2 text-gray-400" />
            <input
              type="text"
              placeholder="Search tasks..."
              value={filters.search || ''}
              onChange={(e) => handleFilterChange('search', e.target.value)}
              className="w-full pl-10 pr-4 py-2 border border-gray-300
rounded-lg focus:outline-none focus:ring-2 focus:ring-blue-500"
            />
          </div>
```

```
          {/* Status Filter */}
          <select
            value={filters.status || ''}
            onChange={(e) => handleFilterChange('status', e.target.value)}
            className="w-full px-4 py-2 border border-gray-300 rounded-lg
focus:outline-none focus:ring-2 focus:ring-blue-500"
          >
            <option value="">All Statuses</option>
            <option value="pending">Pending</option>
            <option value="in-progress">In Progress</option>
            <option value="completed">Completed</option>
          </select>

          {/* Priority Filter */}
          <select
            value={filters.priority || ''}
            onChange={(e) => handleFilterChange('priority', e.target.value)}
            className="w-full px-4 py-2 border border-gray-300 rounded-lg
focus:outline-none focus:ring-2 focus:ring-blue-500"
          >
            <option value="">All Priorities</option>
            <option value="high">High</option>
            <option value="medium">Medium</option>
            <option value="low">Low</option>
          </select>

          {/* Category Filter */}
          <select
            value={filters.category || ''}
            onChange={(e) => handleFilterChange('category', e.target.value)}
            className="w-full px-4 py-2 border border-gray-300 rounded-lg
focus:outline-none focus:ring-2 focus:ring-blue-500"
          >
            <option value="">All Categories</option>
            <option value="work">Work</option>
            <option value="personal">Personal</option>
            <option value="health">Health</option>
            <option value="learning">Learning</option>
            <option value="other">Other</option>
          </select>
        </div>

        <div className="flex justify-between items-center">
          <p className="text-sm text-gray-600">
            Showing {filteredTasks.length} of {tasks.length} tasks
          </p>
          <button
            onClick={clearFilters}
            className="text-blue-600 hover:text-blue-800 text-sm font-medium
transition-colors duration-200"
```

```
          >
            Clear Filters
          </button>
        </div>
      </div>

      {/* Tasks Grid */}
      {filteredTasks.length === 0 ? (
        <div className="bg-white rounded-lg shadow-md p-8 text-center">
          <p className="text-gray-500 mb-4">
            {tasks.length === 0 ? 'No tasks found. Create your first task!' :
'No tasks match your current filters.'}
          </p>
          {tasks.length === 0 && (
            <Link
              to="/add-task"
              className="inline-flex items-center px-4 py-2 bg-blue-600 text-
white rounded-lg hover:bg-blue-700 transition-colors duration-200"
            >
              <Plus size={18} className="mr-2" />
              Create First Task
            </Link>
          )}
        </div>
      ) : (
        <div className="grid grid-cols-1 lg:grid-cols-2 xl:grid-cols-3 gap-
6">
          {filteredTasks.map(task => (
            <TaskCard
              key={task.id}
              task={task}
              onDelete={deleteTask}
              onStatusChange={handleStatusChange}
            />
          ))}
        </div>
      )}
    </div>
  );
};
```

Add Task Page

```
// src/pages/AddTask.tsx
import React from 'react';
import { useTasks } from '../hooks/useTasks';
import { TaskForm } from '../components/tasks/TaskForm';
import { TaskFormData } from '../types';

export const AddTask: React.FC = () => {
  const { addTask } = useTasks();
```

```
  const handleSubmit = (data: TaskFormData) => {
    addTask(data);
  };

  return (
    <div className="container mx-auto px-4 py-8">
      <div className="mb-8">
        <h1 className="text-3xl font-bold text-gray-800 mb-2">Create New
Task</h1>
        <p className="text-gray-600">Add a new task to your list and stay
organized.</p>
      </div>

      <TaskForm onSubmit={handleSubmit} submitButtonText="Create Task" />
    </div>
  );
};
```

Edit Task Page

```
// src/pages/EditTask.tsx
import React from 'react';
import { useParams, Navigate } from 'react-router-dom';
import { useTasks } from '../hooks/useTasks';
import { TaskForm } from '../components/tasks/TaskForm';
import { TaskFormData } from '../types';

export const EditTask: React.FC = () => {
  const { id } = useParams<{ id: string }>();
  const { getTaskById, updateTask } = useTasks();

  if (!id) {
    return <Navigate to="/tasks" replace />;
  }

  const task = getTaskById(id);

  if (!task) {
    return (
      <div className="container mx-auto px-4 py-8">
        <div className="bg-white rounded-lg shadow-md p-8 text-center">
          <h1 className="text-2xl font-bold text-gray-800 mb-4">Task Not
Found</h1>
          <p className="text-gray-600 mb-4">The task you're looking for
doesn't exist.</p>
          <Navigate to="/tasks" replace />
        </div>
      </div>
    );
```

```
  }

  const handleSubmit = (data: TaskFormData) => {
    updateTask(id, data);
  };

  return (
    <div className="container mx-auto px-4 py-8">
      <div className="mb-8">
        <h1 className="text-3xl font-bold text-gray-800 mb-2">Edit Task</h1>
        <p className="text-gray-600">Update your task details.</p>
      </div>

      <TaskForm
        initialData={task}
        onSubmit={handleSubmit}
        submitButtonText="Update Task"
      />
    </div>
  );
};
```

Task Details Page

```
// src/pages/TaskDetails.tsx
import React from 'react';
import { useParams, Link, Navigate } from 'react-router-dom';
import { useTasks } from '../hooks/useTasks';
import { Edit, Calendar, Clock, ArrowLeft, Trash2 } from 'lucide-react';
import { formatDate, getPriorityColor, getStatusColor, isOverdue } from
'../utils/taskUtils';

export const TaskDetails: React.FC = () => {
  const { id } = useParams<{ id: string }>();
  const { getTaskById, updateTask, deleteTask } = useTasks();

  if (!id) {
    return <Navigate to="/tasks" replace />;
  }

  const task = getTaskById(id);

  if (!task) {
    return (
      <div className="container mx-auto px-4 py-8">
        <div className="bg-white rounded-lg shadow-md p-8 text-center">
          <h1 className="text-2xl font-bold text-gray-800 mb-4">Task Not
Found</h1>
          <p className="text-gray-600 mb-4">The task you're looking for
doesn't exist.</p>
```

```
          <Navigate to="/tasks" replace />
        </div>
      </div>
    );
  }

  const handleStatusChange = (e: React.ChangeEvent<HTMLSelectElement>) => {
    updateTask(task.id, { status: e.target.value as any });
  };

  const handleDelete = () => {
    if (window.confirm('Are you sure you want to delete this task?')) {
      deleteTask(task.id);
      // Navigate will be handled by the state change
    }
  };

  return (
    <div className="container mx-auto px-4 py-8">
      {/* Header */}
      <div className="mb-8">
        <Link
          to="/tasks"
          className="inline-flex items-center text-blue-600 hover:text-blue-
800 transition-colors duration-200 mb-4"
        >
          <ArrowLeft size={18} className="mr-2" />
          Back to Tasks
        </Link>
        <div className="flex justify-between items-start">
          <div>
            <h1 className="text-3xl font-bold text-gray-800 mb-
2">{task.title}</h1>
            <div className="flex flex-wrap gap-2">
              <span className={`px-3 py-1 rounded-full text-sm font-medium
${getPriorityColor(task.priority)}`}>
                {task.priority.charAt(0).toUpperCase() +
task.priority.slice(1)} Priority
              </span>
              <span className={`px-3 py-1 rounded-full text-sm font-medium
${getStatusColor(task.status)}`}>
                {task.status.replace('-', ' ').replace(/\b\w/g, l =>
l.toUpperCase())}
              </span>
              <span className="px-3 py-1 rounded-full text-sm font-medium
text-purple-600 bg-purple-100">
                {task.category.charAt(0).toUpperCase() +
task.category.slice(1)}
              </span>
            </div>
```

```
          </div>
          <div className="flex space-x-2">
            <Link
              to={`/edit-task/${task.id}`}
              className="inline-flex items-center px-4 py-2 bg-blue-600 text-
white rounded-lg hover:bg-blue-700 transition-colors duration-200"
            >
              <Edit size={18} className="mr-2" />
              Edit
            </Link>
            <button
              onClick={handleDelete}
              className="inline-flex items-center px-4 py-2 bg-red-600 text-
white rounded-lg hover:bg-red-700 transition-colors duration-200"
            >
              <Trash2 size={18} className="mr-2" />
              Delete
            </button>
          </div>
        </div>
      </div>

      <div className="grid grid-cols-1 lg:grid-cols-3 gap-8">
        {/* Main Content */}
        <div className="lg:col-span-2">
          <div className="bg-white rounded-lg shadow-md p-6 mb-6">
            <h2 className="text-xl font-semibold text-gray-800 mb-
4">Description</h2>
            <p className="text-gray-600 leading-
relaxed">{task.description}</p>
          </div>

          {/* Status Update */}
          <div className="bg-white rounded-lg shadow-md p-6">
            <h2 className="text-xl font-semibold text-gray-800 mb-4">Update
Status</h2>
            <select
              value={task.status}
              onChange={handleStatusChange}
              className="w-full px-4 py-2 border border-gray-300 rounded-lg
focus:outline-none focus:ring-2 focus:ring-blue-500"
            >
              <option value="pending">Pending</option>
              <option value="in-progress">In Progress</option>
              <option value="completed">Completed</option>
            </select>
          </div>
        </div>

        {/* Sidebar */}
```

```
        <div className="space-y-6">
          {/* Due Date */}
          {task.dueDate && (
            <div className="bg-white rounded-lg shadow-md p-6">
              <h3 className="text-lg font-semibold text-gray-800 mb-3">Due
Date</h3>
              <div className={`flex items-center space-x-2 ${
                isOverdue(task.dueDate) && task.status !== 'completed'
                  ? 'text-red-600'
                  : 'text-gray-600'
              }`}>
                {isOverdue(task.dueDate) && task.status !== 'completed' ? (
                  <Clock size={20} />
                ) : (
                  <Calendar size={20} />
                )}
                <span className="font-medium">
                  {formatDate(task.dueDate)}
                </span>
              </div>
              {isOverdue(task.dueDate) && task.status !== 'completed' && (
                <p className="text-sm text-red-600 mt-2">This task is
overdue!</p>
              )}
            </div>
          )}

          {/* Task Info */}
          <div className="bg-white rounded-lg shadow-md p-6">
            <h3 className="text-lg font-semibold text-gray-800 mb-3">Task
Information</h3>
            <div className="space-y-3">
              <div>
                <span className="text-sm font-medium text-gray-
500">Created:</span>
                <p className="text-gray-800">{formatDate(task.createdAt)}</p>
              </div>
              <div>
                <span className="text-sm font-medium text-gray-500">Last
Updated:</span>
                <p className="text-gray-800">{formatDate(task.updatedAt)}</p>
              </div>
              <div>
                <span className="text-sm font-medium text-gray-
500">Priority:</span>
                <p className="text-gray-800 capitalize">{task.priority}</p>
              </div>
              <div>
                <span className="text-sm font-medium text-gray-
500">Category:</span>
```

```
                <p className="text-gray-800 capitalize">{task.category}</p>
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
  );
};
```

🚀 Main App Component with Routing

```
// src/App.tsx
import React from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import { Header } from './components/common/Header';
import { Dashboard } from './pages/Dashboard';
import { Tasks } from './pages/Tasks';
import { AddTask } from './pages/AddTask';
import { EditTask } from './pages/EditTask';
import { TaskDetails } from './pages/TaskDetails';

function App() {
  return (
    <Router>
      <div className="min-h-screen bg-gray-50">
        <Header />
        <main>
          <Routes>
            <Route path="/" element={<Dashboard />} />
            <Route path="/tasks" element={<Tasks />} />
            <Route path="/add-task" element={<AddTask />} />
            <Route path="/edit-task/:id" element={<EditTask />} />
            <Route path="/tasks/:id" element={<TaskDetails />} />
          </Routes>
        </main>
      </div>
    </Router>
  );
}

export default App;
```

🎨 Styling with Tailwind CSS

Make sure your `src/index.css` includes Tailwind:

```
/* src/index.css */
@tailwind base;
@tailwind components;
```

```
@tailwind utilities;

/* Custom styles */
.line-clamp-2 {
  display: -webkit-box;
  -webkit-line-clamp: 2;
  -webkit-box-orient: vertical;
  overflow: hidden;
}

/* Smooth transitions */
* {
  transition-property: color, background-color, border-color, text-
decoration-color, fill, stroke;
  transition-timing-function: cubic-bezier(0.4, 0, 0.2, 1);
  transition-duration: 150ms;
}
```

🔄 Key Learning Points Demonstrated

1. React Hook Form Integration

```
// Proper form validation with TypeScript
const {
  register,
  handleSubmit,
  formState: { errors, isSubmitting },
  reset
} = useForm<TaskFormData>({
  defaultValues: {
    title: initialData?.title || '',
    // ... other defaults
  }
});

// Register inputs with validation
{...register('title', {
  required: 'Task title is required',
  minLength: {
    value: 3,
    message: 'Title must be at least 3 characters long'
  }
})}
```

2. Custom Hooks for Logic Separation

```
// useTasks hook encapsulates all task-related logic
export function useTasks() {
  const [tasks, setTasks] = useLocalStorage<Task[]>('tasks', []);

  const addTask = (taskData: Omit<Task, 'id' | 'createdAt' | 'updatedAt'>) =>
```

```
{
    // Implementation
  };

  return { tasks, addTask, updateTask, deleteTask, stats };
}
```

3. TypeScript Best Practices

```
// Proper typing for component props
interface TaskCardProps {
  task: Task;
  onDelete: (id: string) => void;
  onStatusChange: (id: string, status: Task['status']) => void;
}

// Using utility types
type TaskFormData = Omit<Task, 'id' | 'createdAt' | 'updatedAt' | 'status'>;
```

4. State Management Patterns

```
// Proper state updates with functional updates
setTasks(prev =>
  prev.map(task =>
    task.id === id
      ? { ...task, ...updates, updatedAt: new Date().toISOString() }
      : task
  )
);
```

🚨 Common Pitfalls & Solutions

❌ Pitfall 1: Direct State Mutation

```
// ❌ Wrong - mutating state directly
task.status = 'completed';
setTasks(tasks);

// ✅ Correct - immutable updates
setTasks(prev =>
  prev.map(t => t.id === task.id ? { ...t, status: 'completed' } : t)
);
```

❌ Pitfall 2: Missing Form Validation

```
// ❌ Wrong - no validation
<input {...register('title')} />

// ✅ Correct - proper validation
<input {...register('title', {
  required: 'Title is required',
```

```
  minLength: { value: 3, message: 'Too short' }
})} />
```

❌ Pitfall 3: Incorrect Hook Dependencies

```
// ❌ Wrong - missing dependencies
useEffect(() => {
  updateLocalStorage(tasks);
}, []); // Missing 'tasks' dependency

// ☑ Correct - proper dependencies
useEffect(() => {
  updateLocalStorage(tasks);
}, [tasks]);
```

💡 Pro Tips

Tip 1: Performance optimization with useMemo

```
const filteredTasks = useMemo(() => {
  return tasks.filter(task => {
    // Expensive filtering logic
  });
}, [tasks, filters]); // Only recalculate when dependencies change
```

Tip 2: Form state management

```
// Use React Hook Form's reset for better UX
const handleFormSubmit = async (data: TaskFormData) => {
  await onSubmit(data);
  if (!initialData) {
    reset(); // Clear form only for new tasks
  }
};
```

Tip 3: URL state synchronization

```
// Keep URL in sync with filters
const handleFilterChange = (key: keyof TaskFilters, value: string) => {
  const newFilters = { ...filters, [key]: value || undefined };
  setFilters(newFilters);

  // Update URL
  const params = new URLSearchParams();
  Object.entries(newFilters).forEach(([k, v]) => {
    if (v) params.set(k, v);
  });
  setSearchParams(params);
};
```

🎯 Exercises and Challenges

Exercise 1: Add Search Functionality

Implement real-time search that filters tasks by title and description.

Solution approach:

1. Add search state to Tasks component
2. Use debouncing to avoid excessive filtering
3. Highlight search terms in results

Exercise 2: Add Task Categories

Extend the app to support custom categories.

Requirements:

- Allow users to create custom categories
- Store categories in localStorage
- Add category management page

Exercise 3: Add Task Dependencies

Implement task dependencies where some tasks can block others.

Features to implement:

- Add "depends on" field to tasks
- Visual indication of blocked tasks
- Automatic status updates based on dependencies

Exercise 4: Add Data Export/Import

Allow users to export their tasks as JSON or CSV.

Implementation ideas:

- Add export button to dashboard
- Create import functionality for backup restoration
- Validate imported data structure

Final Challenge: Add Collaboration

Extend the app to support multiple users (simulation only).

Features:

- User profiles
- Task assignment

- Activity history
- Comments on tasks

🎉 Congratulations!

You've successfully built a complete React application that demonstrates all the key concepts from this course:

- ✅ **Component composition** with reusable UI components
- ✅ **State management** with hooks and custom hooks
- ✅ **Form handling** with React Hook Form and validation
- ✅ **Routing** with React Router for navigation
- ✅ **TypeScript integration** with proper typing
- ✅ **Data persistence** with localStorage
- ✅ **Performance optimization** with useMemo and proper re-rendering
- ✅ **Error handling** and user feedback
- ✅ **Modern React patterns** and best practices

🚀 Next Steps

Now that you've completed this course, here are some suggestions for further learning:

1. **State Management Libraries**: Learn Redux Toolkit or Zustand for complex state management
2. **Backend Integration**: Connect your app to a real API using Axios or React Query
3. **Testing**: Add unit and integration tests with Jest and React Testing Library
4. **Performance**: Learn about React.memo, useCallback, and advanced optimization
5. **Advanced Patterns**: Explore compound components, render props, and custom hook patterns
6. **Deployment**: Deploy your app to Vercel, Netlify, or AWS

You now have a solid foundation in React development with TypeScript. Keep building, keep learning, and most importantly, keep coding! 🚀

Happy coding! 💻✨