

Lab 5: React Advanced — Practical Exercises

Overview

These exercises are designed to reinforce the advanced concepts covered in Module 5, including complex state management, performance optimization, advanced design patterns, and integration testing.

Part 1: Complex State Management

Objective: Transition from imperative state manipulation to deterministic state transitions using `useReducer` and Redux Toolkit.

Exercise 1.1: The Fetch Machine (`useReducer`)

Context: `useState` is often insufficient for complex state logic where multiple sub-values change together or depend on previous states. The `useReducer` hook is preferred for these scenarios to avoid "impossible states" (e.g., loading and error both being true).

Scenario:

You have a component `UserProfile` that fetches user data. Currently, it uses three separate `useState` hooks (`isLoading`, `error`, `data`).

Task:

1. Refactor the component to use `useReducer`.
2. Define a reducer function that handles the following action types:
 - o 'FETCH_INIT'
 - o 'FETCH_SUCCESS'
 - o 'FETCH_FAILURE'
3. **Challenge:** Implement a Finite State Machine pattern where the reducer prevents invalid transitions (e.g., preventing a transition to `FETCH_SUCCESS` if the current state is not `LOADING`).

Starter Logic:

```
// Current problematic state
const = useState(null);
const [loading, setLoading] = useState(false);
const [error, setError] = useState(null);

// Refactor to:
const initialState = { status: 'idle', data: null, error: null };
// status can be: 'idle' | 'loading' | 'resolved' | 'rejected'
```

Exercise 1.2: The Global Store (Redux Toolkit)

Context: Redux Toolkit (RTK) simplifies Redux setup by providing tools like configureStore and createSlice, which automatically generate action creators and handle immutability.

Scenario:

Your application needs a global "Shopping Cart" feature.

Task:

1. Use configureStore to set up the Redux store.
2. Use createSlice to define a cartSlice.
 - o **State:** { items:, totalAmount: 0 }
 - o **Reducers:**
 - addItem: Handles adding a product or incrementing quantity if it exists.
 - removeItem: Removes an item or decrements quantity.
 - clearCart: Resets the state.
3. **Challenge:** Use createSelector (memoized selector) to derive a value selectCartTax that calculates a 10% tax on the totalAmount. Ensure this calculation only runs when totalAmount changes.

Part 2: Performance Engineering

Objective: Diagnose and fix performance bottlenecks using React's optimization hooks and Code Splitting.

Exercise 2.1: The Laggy List (useMemo & React.memo)

Context: React.memo prevents unnecessary re-renders of child components when props haven't changed. useMemo caches the result of expensive calculations.

Scenario:

You are provided with a Dashboard component that renders a LargeList (10,000 items) and has a generic ThemeToggle button. Currently, every time the user toggles the theme (dark/light), the entire list re-sorts and re-renders, causing a visible UI freeze.

Task:

1. Wrap the sorting logic of the list items in useMemo so it only recalculates when the items prop changes, not when the theme changes.⁷
2. Wrap the ListItem child component in React.memo.⁸
3. **Observation:** Use the React DevTools "Profiler" tab to record a session before and after your changes. Quantify the reduction in render time.

Exercise 2.2: Stabilization (useCallback)

Context: Passing inline functions to memoized components breaks optimization because a new function reference is created on every render. useCallback stabilizes these references.

Scenario:

In the previous exercise, passing an onDelete function to the ListItem breaks the React.memo optimization because the function reference changes on every parent render.

Task:

1. Wrap the handleDelete function in the parent component using useCallback.
2. Verify via console.log inside the ListItem that it no longer re-renders when the parent's unrelated state (theme) changes.

Exercise 2.3: Route-Based Code Splitting

Context: Code splitting allows you to split your bundle into smaller chunks, loading them only when needed. This is typically achieved using React.lazy and Suspense.

Scenario:

The application's initial bundle size is too large (5MB). The AdminPanel page is rarely used but includes heavy charting libraries.

Task:

1. Refactor the App.js router. Change the static import of AdminPanel to a dynamic import using React.lazy.
2. Wrap the route in a Suspense component.
3. Create a LoadingSpinner component and pass it to the fallback prop of Suspense.

Part 3: Advanced Design Patterns

Objective: Build reusable, flexible component libraries using Compound Components and Portals.

Exercise 3.1: The Compound Tabs Component

Context: The Compound Component pattern (like `<select>` and `<option>`) allows components to share implicit state using Context, providing a flexible API for consumers.⁹

Scenario:

You need to build a reusable `<Tabs>` component that allows developers to arrange tab headers and panels in any order they wish.

Task:

1. Create a `TabsContext` using `createContext`.
2. Build the Parent `<Tabs>` component that holds the state (`activeTabIndex`) and provides it via Context.¹⁰
3. Build child components: `<Tabs.List>`, `<Tabs.Tab>`, and `<Tabs.Panel>`.
4. **Requirement:** The consumer of your component should be able to write code like this:

```
<Tabs defaultIndex={0}>
  <Tabs.List>
    <Tabs.Tab index={0}>React</Tabs.Tab>
    <Tabs.Tab index={1}>Redux</Tabs.Tab>
  </Tabs.List>
  <div className="divider"></div> /* Custom markup allowed here */
  <Tabs.Panel index={0}>React is a library...</Tabs.Panel>
  <Tabs.Panel index={1}>Redux is a store...</Tabs.Panel>
</Tabs>
```

Exercise 3.2: The "Trapdoor" Modal (Portals)

Context: `createPortal` lets you render a child component into a DOM node that exists outside the DOM hierarchy of the parent, which is essential for modals to avoid CSS clipping.

Scenario:

A modal dialog inside a card component is being clipped because the card has `overflow: hidden` CSS.

Task:

1. Create a `Modal` component that uses `ReactDOM.createPortal`.

2. Render the modal content into document.body (or a specific #modal-root div in index.html), escaping the parent's CSS stacking context.
 3. **Event Bubbling Challenge:** Add an onClick listener to the *parent* div inside the main App tree. Verify that clicking the button inside the Portal (which is physically outside the div) still triggers the parent's click listener due to React's synthetic event bubbling.
-

Part 4: Testing Strategies

Objective: Write resilient tests using Jest and React Testing Library (RTL) that focus on user behavior.

Exercise 4.1: Integration Testing a Form

Context: Integration tests check how multiple units work together. RTL encourages testing user interactions (like clicking and typing) rather than implementation details.

Scenario:

You have a LoginForm component with Email/Password inputs and a Submit button. When submitted, it calls an API and displays a success message.

Task:

1. **Arrange:** Render the <LoginForm />.
2. **Act:**
 - Use userEvent.type to fill in the email and password.
 - Use userEvent.click to submit the form.
3. **Assert:**
 - Mock the API request (using jest.mock or MSW).
 - Use await screen.findByText(/welcome back/i) to assert the success message appears.
 - *Anti-Pattern Check:* Do not test the internal state of the component. Test only what the user sees.

Exercise 4.2: Testing Error Boundaries

Context: Error Boundaries are class components that catch JavaScript errors in their child component tree. The react-error-boundary library provides a functional wrapper for this.

Scenario:

You have implemented an ErrorBoundary using react-error-boundary.

Task:

1. Create a "Bomb" component that throw new Error('Boom!') when rendered.

2. Write a test that renders the Bomb inside your ErrorBoundary.
3. Assert that the application does not crash and that the Fallback UI (e.g., "Something went wrong") is visible in the DOM.
4. Ensure the console error is silenced during the test execution to keep test logs clean (optional).