React Advanced

A Complete 7-Chapter Course

Topics covered:

State Management (Zustand/Redux Toolkit) · React Query · Code Splitting & Suspense

Testing (Jest/Vitest + RTL) · Advanced Patterns · Performance Profiling · Next.js & SSR

Exercises: 21 hands-on challenges with sample solutions

Format: A4 · Dark-theme code examples · Quick-reference tables

Table of Contents

CHAPTER 1 OF 7

State Management Beyond Context

COURSE 3 · CH 1

State Management Beyond Context

Welcome to Advanced. Zustand and Redux Toolkit — what they solve that Context + useReducer eventually can't

Context plus `useReducer` — used in the Shopping Cart project and formalized in Intermediate Chapters 2–3 — genuinely works for global state, and Chapter 7's `useMemo` fix solved its biggest re-render problem. It still has real limits at scale: every piece of global state generally needs its own Provider wrapping the app, and even with memoization, splitting state cleanly across many unrelated concerns gets unwieldy. **Zustand** and **Redux Toolkit** are the two most common dedicated answers to that — different philosophies, both still built around the same core idea: state living outside any one component, with components subscribing to the pieces they need.

Zustand — Minimal Global State

```
// npm install zustand
import { create } from "zustand";

const useCounterStore = create((set) => ({
  count: 0,
  increment: () => set((state) => ({ count: state.count + 1 })),
  reset: () => set({ count: 0 }),
}));

// usage — anywhere, no Provider needed at all
function Counter() {
  const count = useCounterStore((state) => state.count);
  const increment = useCounterStore((state) => state.increment);
  return <button onClick={increment}>{count}</button>;
}
```

`create()` builds a store and a hook for reading it, in one step — the store itself lives entirely outside the React component tree, so there's no `<Provider>` wrapping anything, unlike every Context example so far. Each call to `useCounterStore(selector)` subscribes only to the specific piece of state that selector returns — `Counter` only re-renders when `count` itself changes, not when some unrelated piece of the same store

changes, solving Context's all-or-nothing re-render problem directly rather than needing the `useMemo` workaround from Chapter 7.

Rebuilding the Shopping Cart with Zustand

```
const useCartStore = create((set) => ({
  items: [],
  addToCart: (product) =>
    set((state) => {
      const existing = state.items.find((i) => i.id === product.id);
      if (existing) {
        return {
          items: state.items.map((i) =>
            i.id === product.id ? { ...i, quantity: i.quantity + 1 } : i
          ),
        };
      }
      return { items: [...state.items, { ...product, quantity: 1 }] };
    }),
  removeFromCart: (id) =>
    set((state) => ({ items: state.items.filter((i) => i.id !== id) })),
}));
```

The duplicate-entry-check logic from Project 6's `addToCart` is identical here — Zustand changes *where* state lives and how components subscribe to it, not the immutable-update rules underneath. Any component anywhere can call `useCartStore((s) => s.items)` or `useCartStore((s) => s.addToCart)` directly — no `CartProvider` wrapping the app, no `useContext` call.

Redux Toolkit — More Structure, More Ceremony

```jsx
// npm install @reduxjs/toolkit react-redux
import { createSlice, configureStore } from "@reduxjs/toolkit";
import { Provider, useSelector, useDispatch } from "react-redux";

const counterSlice = createSlice({
  name: "counter",
  initialState: { count: 0 },
  reducers: {
    increment: (state) => { state.count += 1; },  // "mutation" here is safe — Redux Toolkit uses Immer
  },
});

const store = configureStore({ reducer: { counter: counterSlice.reducer } });

// still needs a Provider, wrapping the app once — same shape as Context
<Provider store={store}><App /></Provider>

// usage
const count = useSelector((state) => state.counter.count);
const dispatch = useDispatch();
dispatch(counterSlice.actions.increment());
```

Redux Toolkit follows the same `dispatch`/action/reducer shape as `useReducer` from Intermediate Chapter 3, scaled up with "slices" for organizing many pieces of state, a single combined store, and (unlike a hand-written reducer) the ability to write code that *looks* like direct mutation inside a reducer safely — Redux Toolkit uses a library called Immer underneath to translate that into a proper immutable update automatically. It does still need a `<Provider>` wrapping the app, much like Context.

NEITHER OF THESE IS THE DEFAULT CHOICE

Both tools solve real problems at a certain scale — many independent pieces of global state, performance issues from Context re-renders that `useMemo` can't fully solve, or a team that specifically wants Redux's strict structure and devtools ecosystem. If `useReducer` + Context (with the Chapter 7 `useMemo` fix) is working fine for an app's actual needs, that's a completely reasonable place to stop — neither Zustand nor Redux Toolkit need to be reached for "just in case."

	CONTEXT + USEREDUCER	ZUSTAND	REDUX TOOLKIT
Needs a Provider?	Yes	No	Yes
Selective re-renders?	Manual (useMemo)	Built in	Built in (useSelector)
Setup ceremony	Low	Very low	Higher
Ecosystem / devtools	None built-in	Small, growing	Large, mature
Best fit	Small-to-medium global state	Most apps wanting simplicity	Large teams, strict conventions

Coding Challenges

CHALLENGE 1

Build a Zustand counter store with increment, decrement, and reset actions, and use it from two completely unrelated components, confirming both stay in sync with no Provider anywhere in the app.

View solution

CHALLENGE 2

Rebuild Project 6's shopping cart logic (items, addToCart with the duplicate-check behavior, removeFromCart) as a Zustand store, and use it from a product list component and a separate cart-display component.

View solution

CHALLENGE 3

Build the same counter from Challenge 1 using Redux Toolkit instead (createSlice, configureStore, a Provider, useSelector, useDispatch), and compare the amount of code each approach needed for identical behavior.

View solution

CHAPTER 1 QUICK REFERENCE

- **Zustand** — `create()` builds a store + hook; no Provider; selective subscriptions built in
- **Redux Toolkit** — `createSlice` / `configureStore`; needs a `Provider`; mature ecosystem/devtools
- Both still rely on the same **immutable-update rules** from Fundamentals Ch 3 underneath (Redux Toolkit just hides it via Immer)
- Neither replaces Context + useReducer by default — reach for them when scale genuinely demands it
- **Next chapter:** React Query — the same pattern applied specifically to server data, first used informally in Project 7

CHAPTER 2 OF 7

Server State with React Query

COURSE 3 · CH 2

Server State with React Query

The useFetch hook from Intermediate Chapter 6, properly automated — caching, refetching, and race conditions handled for you

Intermediate Chapter 6 built a `useFetch` hook by hand: loading/error state, a manual `response.ok` check, and an ignore-flag to fix a genuine race condition. Project 7's dashboard used **React Query**'s `useQuery` informally to skip writing all of that. This chapter explains exactly what it's doing underneath, and goes further — caching, automatic refetching, and updating server data with `useMutation`.

Setup, Recapped

```
// npm install @tanstack/react-query
import { QueryClient, QueryClientProvider } from "@tanstack/react-query";

const queryClient = new QueryClient();

function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <Dashboard />
    </QueryClientProvider>
  );
}
```

One `QueryClient`, wrapping the app once — the same Provider shape seen with Context and Redux Toolkit, holding a cache shared by every `useQuery` call anywhere below it.

How queryKey Solves Chapter 6's Race Condition — Automatically

```
import { useQuery } from "@tanstack/react-query";

function ProductDetail({ id }) {
  const { data, isLoading, isError } = useQuery({
    queryKey: ["product", id],
    queryFn: () => fetch(`/api/products/${id}`).then((r) => r.json()),
  });

  if (isLoading) return <p>Loading...</p>;
  if (isError) return <p>Something went wrong.</p>;
  return <h2>{data.name}</h2>;
}
```

Including `id` directly inside `queryKey` — `["product", id]` — means React Query treats every distinct `id` as a *separate* cache entry. Switching rapidly between two ids no longer risks the stale-overwrite bug from Chapter 6's challenge: each id's request and result are tracked independently by key, so a slow response for an old id can never land on top of a newer id's already-displayed data. No ignore flag, no `AbortController`, written by hand at all — the keying scheme itself prevents the problem from Chapter 6 entirely.

Caching in Practice

The first component to call `useQuery({ queryKey: ["product", 5], ... })` triggers the actual fetch; if a second, completely unrelated component requests the exact same key later, React Query returns the already-cached result instantly, without firing a second network request at all — this is what made Project 7's badge-plus-detail-page style dashboard so much simpler than hand-rolling the same sharing logic with `useFetch`.

```
useQuery({
  queryKey: ["coins"],
  queryFn: fetchCoins,
  staleTime: 30000,        // treat cached data as fresh for 30 seconds
  refetchInterval: 60000, // also refetch automatically every 60 seconds
});
```

`staleTime` controls how long cached data is considered "still good enough" before React Query will bother refetching it on the next render of a component using that key; `refetchInterval` (used informally in Project 7) is for actively polling data that changes on its own over time, like live prices.

useMutation — Changing Server Data

```jsx
import { useMutation, useQueryClient } from "@tanstack/react-query";

function AddTodoForm() {
  const queryClient = useQueryClient();

  const mutation = useMutation({
    mutationFn: (newTodo) =>
      fetch("/api/todos", { method: "POST", body: JSON.stringify(newTodo) }),
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: ["todos"] });
    },
  });

  return (
    <button onClick={() => mutation.mutate({ text: "New todo" })}>
      Add
    </button>
  );
}
```

`useQuery` is for **reading** server data; `useMutation` is for **changing** it — a POST/PUT/DELETE request triggered by calling `mutation.mutate(...)`, typically from an event handler rather than automatically on render. `onSuccess`'s call to `queryClient.invalidateQueries({ queryKey: ["todos"] })` tells React Query that any cached data under that key is now out of date, triggering every component reading `["todos"]` to refetch automatically — the standard way a mutation keeps the rest of the UI in sync with what just changed on the server.

> **DON'T COPY QUERY DATA INTO USESTATE**
>
> Copying `data` from `useQuery` into a separate `useState` "just to have a local copy" reintroduces exactly the staleness problems this whole chapter exists to avoid — that local copy won't update when the cache refreshes or a mutation invalidates it. Treat `useQuery`'s return value as the live source of truth directly; if a derived value is needed, compute it inline or with `useMemo` (Intermediate Chapter 7), rather than storing a snapshot.

HAND-ROLLED (CH 6)	REACT QUERY EQUIVALENT
`useState + useEffect + fetch`	`useQuery`
`Manual status state`	`isLoading` / `isError` / `isSuccess`
`Ignore-flag race condition fix`	Per-key caching, automatic
`Manually re-calling setData after a POST`	`useMutation` + `invalidateQueries`

Coding Challenges

CHALLENGE 1

Build a component fetching a list from any free public API using useQuery, handling isLoading and isError, and rendering the list on success.

View solution

CHALLENGE 2

Recreate Intermediate Chapter 6 Challenge 2's race-condition scenario (two buttons switching between a slow and a fast id) using useQuery with the id included in queryKey, and confirm the stale-overwrite bug no longer happens — with no ignore flag written by hand.

View solution

CHALLENGE 3

Build a small todo list backed by useQuery (fetching the list) and useMutation (adding a new todo), with onSuccess calling invalidateQueries so the list refetches and shows the new item automatically.

View solution

CHAPTER 2 QUICK REFERENCE

- **QueryClientProvider** wraps the app; **useQuery** reads, **useMutation** writes
- **queryKey** identifies a cache entry — including variables (like an id) in it keys each one separately
- Two components using the same `queryKey` share one cached result — no duplicate fetch
- **staleTime** / **refetchInterval** control how fresh cached data needs to be, and automatic polling
- **invalidateQueries** after a mutation tells React Query to refetch affected data automatically
- Never copy `useQuery`'s data into a separate `useState` — it reintroduces staleness
- **Next chapter:** code splitting and Suspense — loading parts of an app only when actually needed

CHAPTER 3 OF 7

Code Splitting and Suspense

COURSE 3 · CH 3

Code Splitting and Suspense

Loading parts of an app only when they're actually needed, instead of all at once on first load

Every component built so far has been bundled into the app's JavaScript from the very first page load, regardless of whether it's ever used in a given visit. As an app grows — more pages, more rarely-used features, larger third-party libraries (a charting library, a rich text editor) — that single bundle keeps growing too, even for a user who only ever visits one page. **Code splitting** breaks the bundle into smaller pieces, loaded on demand; **Suspense** is what shows something reasonable on screen while a piece is still being fetched.

React.lazy — Loading a Component on Demand

```
import { lazy, Suspense } from "react";

const SettingsPage = lazy(() => import("./SettingsPage"));

function App() {
  return (
    <Suspense fallback={<p>Loading...</p>}>
      <SettingsPage />
    </Suspense>
  );
}
```

`lazy(() => import("./SettingsPage"))` doesn't load `SettingsPage`'s code at all until the moment it's actually about to be rendered for the first time — the dynamic `import()` call kicks off a separate network request for just that component's code, fetched as its own small bundle ("chunk") rather than being part of the main one. While that chunk is loading, the nearest wrapping `<Suspense fallback={...}>` shows its `fallback` instead of the real component, swapping over automatically the instant loading finishes.

REACT.LAZY NEEDS A DEFAULT EXPORT

`lazy()` expects the dynamically imported module to have a `default` export — `import("./SettingsPage")` resolves to an object, and `lazy` specifically looks for that object's `.default` property. A module exporting only a

named export (`export function SettingsPage() {...}`, no `export default`) needs a small wrapper module re-exporting it as default, or needs to be changed to a default export directly.

Route-Based Splitting — The Most Common Use Case

```
import { lazy, Suspense } from "react";
import { BrowserRouter, Routes, Route } from "react-router-dom";

const HomePage = lazy(() => import("./pages/HomePage"));
const SettingsPage = lazy(() => import("./pages/SettingsPage"));

function App() {
  return (
    <BrowserRouter>
      <Suspense fallback={<p>Loading page...</p>}>
        <Routes>
          <Route path="/" element={<HomePage />} />
          <Route path="/settings" element={<SettingsPage />} />
        </Routes>
      </Suspense>
    </BrowserRouter>
  );
}
```

Pairing `lazy` with the routes from Intermediate Chapter 5 is the single most common pattern: a user visiting `/` only ever downloads `HomePage`'s code, `SettingsPage`'s chunk never loads at all unless that user actually navigates to `/settings`. One `Suspense` boundary wrapping the whole `Routes` block is usually enough — every route shares the same fallback while its own page's chunk loads.

Loading a Rarely-Used Feature on Click

```jsx
const AnalyticsChart = lazy(() => import("./AnalyticsChart"));

function Dashboard() {
  const [showChart, setShowChart] = useState(false);

  return (
    <div>
      <button onClick={() => setShowChart(true)}>Show analytics</button>
      {showChart && (
        <Suspense fallback={<p>Loading chart...</p>}>
          <AnalyticsChart />
        </Suspense>
      )}
    </div>
  );
}
```

Here, `AnalyticsChart`'s code (likely a sizeable charting library) doesn't load on the initial page visit at all — only once `showChart` becomes `true`, the conditional rendering from Fundamentals Chapter 5 means React only tries to render `<AnalyticsChart />` (triggering its chunk to load) at the exact moment a user actually wants it.

Nesting Suspense Boundaries

A single component can be wrapped in its own dedicated `Suspense`, separate from a larger one further up — useful when one part of a page should show its own specific loading state rather than blocking everything else on the page behind one shared fallback. There's a real tradeoff either way: fewer, broader boundaries mean simpler fallback UI but more of the page "waits together"; more, narrower boundaries let unrelated parts of a page appear independently, at the cost of more fallback states to design.

DON'T LAZY-LOAD EVERYTHING

Wrapping every small component in `lazy` "just in case" creates a waterfall of tiny network requests, often slower overall than one slightly larger bundle loaded once. Code splitting earns its keep at natural boundaries — whole routes, and genuinely large or rarely-used features (a chart library, a rich text editor, an admin-only panel) — not for ordinary, frequently-used, lightweight components.

SUSPENSE FOR DATA — AN EMERGING PATTERN

Newer versions of React and React Query are extending `Suspense` beyond just component code, letting a data-fetching hook also "suspend" rendering until its data arrives, using the same `Suspense` boundary and fallback shown here. The component-code-splitting pattern covered in this chapter is the stable, well-established baseline either way — worth knowing this direction exists, without needing to chase it immediately.

Coding Challenges

CHALLENGE 1

Build a component lazy-loaded with React.lazy, wrapped in a Suspense boundary with a visible fallback ("Loading..."), and confirm in the browser's network tab that its code arrives as a separate request rather than being part of the main bundle.

View solution

CHALLENGE 2

Build a small multi-page app (using React Router from Intermediate Chapter 5) where every page component is lazy-loaded, with one shared Suspense boundary wrapping the Routes block.

View solution

CHALLENGE 3

Build a Dashboard with a button that, only when clicked, reveals a lazy-loaded component wrapped in its own Suspense boundary — confirming the lazy component's code is not requested at all until the button is actually clicked.

View solution

CHAPTER 3 QUICK REFERENCE

- **lazy(() => import("./X"))** — loads a component's code only when it's first rendered
- **<Suspense fallback={...}>** — shows the fallback while a wrapped lazy component's chunk is loading
- `lazy` requires a **default export** from the imported module
- Most common use: **route-based splitting**, pairing `lazy` with React Router (Intermediate Ch 5)
- Also useful for rarely-used features loaded only on demand (a click, a conditional render)
- Don't lazy-load everything — apply it at route boundaries and genuinely large/rare features, not small everyday components
- **Next chapter:** testing — Jest and React Testing Library

CHAPTER 4 OF 7

Testing — Jest/Vitest and React Testing Library

COURSE 3 · CH 4

Testing — Jest/Vitest and React Testing Library

Confirming components behave correctly without manually clicking through the app to check, every time

Every project so far has been verified by hand — running the app and clicking around to confirm something works. That's fine for small projects, but doesn't scale, and gives nothing to catch a regression introduced weeks later by an unrelated change. **React Testing Library** (paired with a test runner — Jest, or Vite's own Vitest) lets that same kind of verification be written once and run automatically.

JEST VS VITEST

Jest is the long-standing standard test runner for JavaScript/React projects (used by Next.js and older Create React App setups); **Vitest** is Vite's own equivalent, built to be a near drop-in replacement with an almost identical API (`describe`, `test`, `expect` all work the same way). Since every project in this course has used Vite, Vitest is the natural fit going forward — everything in this chapter applies equally to either, since React Testing Library itself doesn't care which one is running it.

The Core Philosophy

React Testing Library is built around one guiding rule: **test what a user actually sees and does**, not a component's internal implementation. That means querying for visible text, labels, and accessible roles — never reaching into a component's state, props, or internal function calls directly. A test written this way keeps passing through a refactor that changes *how* a component works internally, as long as it still behaves the same way from the outside — exactly the kind of test that's actually worth having.

A First Test

```jsx
// Greeting.jsx
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}

// Greeting.test.jsx
import { render, screen } from "@testing-library/react";
import { describe, test, expect } from "vitest";
import Greeting from "./Greeting";

describe("Greeting", () => {
  test("renders the given name", () => {
    render(<Greeting name="Philip" />);
    expect(screen.getByText("Hello, Philip!")).toBeInTheDocument();
  });
});
```

`render(<Greeting name="Philip" />)` renders the component into a virtual DOM the test can inspect; `screen.getByText(...)` searches that rendered output for matching text, throwing immediately if nothing matches; `expect(...).toBeInTheDocument()` is the actual assertion — confirming the element was actually found.

Simulating User Interaction

```jsx
// Counter.jsx — the same component from Fundamentals Chapter 3

// Counter.test.jsx
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import Counter from "./Counter";

test("increments the count when +1 is clicked", async () => {
  const user = userEvent.setup();
  render(<Counter />);

  await user.click(screen.getByRole("button", { name: "+1" }));

  expect(screen.getByText("Count: 1")).toBeInTheDocument();
});
```

`userEvent` simulates real user interaction more faithfully than directly firing a raw DOM event — clicking, typing, and focusing all go through the same library, recommended over the lower-level `fireEvent` for exactly that reason. `getByRole("button", { name: "+1" })` finds the button by its accessible role and visible label — the same way a screen reader or a person scanning the page would identify it, rather than relying on a CSS class or test-specific attribute.

Testing a Form

```
test("calls onSubmit with the typed value", async () => {
  const user = userEvent.setup();
  const handleSubmit = vi.fn();  // jest.fn() in Jest

  render(<SearchForm onSubmit={handleSubmit} />);

  await user.type(screen.getByRole("textbox"), "react");
  await user.click(screen.getByRole("button", { name: "Search" }));

  expect(handleSubmit).toHaveBeenCalledWith("react");
});
```

`vi.fn()` (or `jest.fn()`) creates a **mock function** — a fake function the test can pass in as a prop and later inspect, asking "was this called, and with what?" without needing a real implementation behind it. Passing a mock as `onSubmit` lets the test confirm `SearchForm` calls it correctly with the typed value, without needing the component to actually be connected to a real API anywhere.

DON'T TEST IMPLEMENTATION DETAILS

Reaching for a component's internal state, calling its functions directly, or asserting on a CSS class name couples a test to *how* a component is built rather than *what it does* — exactly the kind of test that breaks on a harmless refactor (renaming a variable, switching from `useState` to `useReducer`) even though the component's actual behavior never changed. Querying by role, label, and visible text — the same things a real user would see and interact with — keeps a test meaningful regardless of what changes underneath.

QUERY	PREFER IT WHEN…
`getByRole`	Almost always the first choice — matches how assistive tech sees the page
`getByLabelText`	Form fields specifically
`getByText`	Plain visible text with no clearer role/label
`getByTestId`	Last resort — nothing else can uniquely identify the element

Coding Challenges

CHALLENGE 1

Write a test for a Greeting component (accepting a name prop) confirming it renders the expected text for at least two different names.

View solution

CHALLENGE 2

Write a test for a Counter component (from Fundamentals Chapter 3) using userEvent to click the +1 button twice, asserting the displayed count updates correctly each time.

View solution

CHALLENGE 3

Write a test for a controlled search input + submit button, using userEvent.type to type a value and a mock function (vi.fn()/jest.fn()) passed as an onSubmit prop, asserting it was called with the typed value.

View solution

CHAPTER 4 QUICK REFERENCE

- **render()** mounts a component for testing; **screen.getByX** queries its rendered output
- **userEvent** simulates real clicks/typing — preferred over the lower-level `fireEvent`
- **vi.fn() / jest.fn()** — a mock function, for asserting it was called (and with what)
- Query priority: **getByRole** > getByLabelText > getByText > getByTestId (last resort)
- Test what a user sees/does — never a component's internal state or implementation details
- Vitest (Vite's test runner) and Jest share nearly identical syntax — everything here applies to both
- **Next chapter:** advanced patterns — compound components, render props, higher-order components

CHAPTER 5 OF 7

Advanced Patterns — Compound Components, Render Props, HOCs

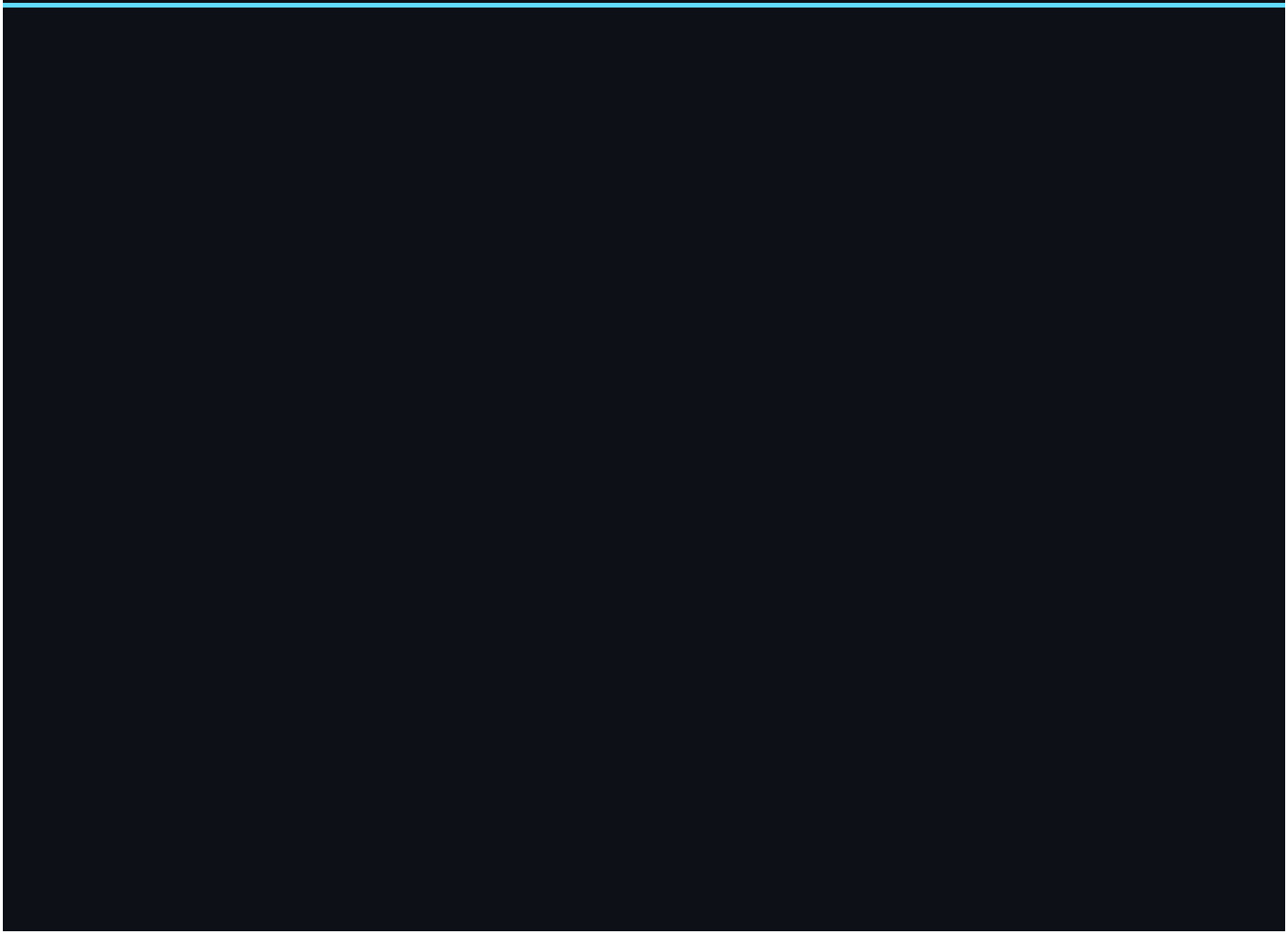
COURSE 3 · CH 5

Advanced Patterns — Compound Components, Render Props, HOCs

Three classic approaches to flexible component APIs — one still common, two largely superseded by hooks

These three patterns predate (in two cases) or coexist with the hooks covered throughout this course, and component libraries you'll use professionally still rely on all three in places. Worth recognizing each one on sight, even where a custom hook (Intermediate Chapter 4) would now be the more natural choice for new code.

Compound Components — Sharing State Implicitly



```jsx
const TabsContext = createContext(null);

function Tabs({ children }) {
  const [activeTab, setActiveTab] = useState(0);
  return <TabsContext.Provider value={{ activeTab, setActiveTab }}>{children}</TabsContext.Provider>;
}

function Tab({ index, children }) {
  const { activeTab, setActiveTab } = useContext(TabsContext);
  return (
    <button onClick={() => setActiveTab(index)} style={{ fontWeight: activeTab === index ? "bold" : "nor
      {children}
    </button>
  );
}

function Panel({ index, children }) {
  const { activeTab } = useContext(TabsContext);
  return activeTab === index ? <div>{children}</div> : null;
}

Tabs.Tab = Tab;
Tabs.Panel = Panel;

// usage — looks like one cohesive widget, no explicit state passed by the caller
<Tabs>
  <Tabs.Tab index={0}>Profile</Tabs.Tab>
  <Tabs.Tab index={1}>Settings</Tabs.Tab>
  <Tabs.Panel index={0}>Profile content</Tabs.Panel>
  <Tabs.Panel index={1}>Settings content</Tabs.Panel>
</Tabs>
```

A **compound component** is a set of components designed to be used together, internally coordinating via Context the way `Tabs` does here — the consumer never sees `activeTab` or passes it anywhere explicitly; `Tabs`, `Tabs.Tab`, and `Tabs.Panel` all share it transparently through context set up entirely inside the family. Attaching `Tab` and `Panel` as properties on `Tabs` (`Tabs.Tab = Tab`) is purely a naming/organization convention, making clear which pieces belong together without a separate import for each. This is genuinely still the standard approach for multi-part UI widgets in real component libraries (tabs, accordions, dropdown menus).

Render Props — Letting the Consumer Control Rendering

```jsx
function Toggle({ children }) {
  const [on, setOn] = useState(false);
  const toggle = () => setOn((v) => !v);

  return children({ on, toggle });
}

// usage — the caller decides exactly what to render, Toggle only owns the behavior
<Toggle>
  {({ on, toggle }) => (
    <button onClick={toggle}>{on ? "ON" : "OFF"}</button>
  )}
</Toggle>
```

Here, `children` isn't JSX at all — it's a **function**, called with whatever data/behavior `Toggle` wants to expose, returning the JSX to actually render. This particular shape ("children as a function") is the most common variant of the render props pattern; the same idea also appears as a separately-named prop (`render={({ on, toggle }) => ...}`) rather than `children` specifically. Either way, `Toggle` owns the boolean state and the toggle logic, while the caller decides completely freely what that state should look like on screen — a button, a checkbox, anything.

> **A CUSTOM HOOK USUALLY DOES THE SAME JOB MORE SIMPLY NOW**
>
> The `Toggle` example above is solving the exact same problem as Intermediate Chapter 4's `useToggle` hook — reusable toggle behavior, usable from anywhere. A custom hook is generally the more direct modern choice for sharing *behavior* specifically; render props remain genuinely useful when a component needs to share both behavior *and* markup structure together (a list component handling virtualization/scrolling, exposing a render prop per row), which a hook alone can't do.

Higher-Order Components — A Function That Returns a Component

```jsx
function withLoading(Component) {
  return function WithLoadingWrapper({ isLoading, ...props }) {
    if (isLoading) {
      return <p>Loading...</p>;
    }
    return <Component {...props} />;
  };
}

const UserProfileWithLoading = withLoading(UserProfile);

// usage
<UserProfileWithLoading isLoading={isLoading} user={user} />
```

A **higher-order component** is a function taking a component and returning a *new* component with extra behavior layered on — here, `withLoading` intercepts the `isLoading` prop and shows a loading message instead of rendering the wrapped component at all, until it becomes `false`. This pattern predates hooks entirely, and was the standard way to share cross-cutting behavior (loading states, authentication checks, data injection) before `useState`/`useEffect` existed in their current form.

HOCS ARE LARGELY LEGACY — PREFER A CUSTOM HOOK FOR NEW CODE

Almost everything a HOC like `withLoading` does, a custom hook does more directly — no wrapper component, no extra layer in the component tree showing up in debugging tools, and no prop-name collisions between what the HOC injects and what the wrapped component already expects. New code today reaches for a custom hook (Intermediate Ch 4) for this kind of cross-cutting concern almost every time; recognizing the HOC pattern still matters for reading and maintaining existing libraries and codebases that predate hooks.

PATTERN	STILL COMMON TODAY?	MODERN ALTERNATIVE
`Compound components`	Yes — multi-part UI widgets	(itself, still the standard approach)
`Render props`	Occasionally — shared behavior + markup structure	A custom hook, when only behavior needs sharing
`HOCs`	Rarely in new code — mostly legacy	A custom hook, almost always

Coding Challenges

CHALLENGE 1

Build a compound Tabs component family (Tabs, Tabs.Tab, Tabs.Panel) sharing active-tab state via Context internally, with at least three tabs.

View solution

CHALLENGE 2

Build a render-props Toggle component (children as a function exposing { on, toggle }), and use it twice to build two completely different UIs — a button showing ON/OFF, and a checkbox — both reusing the exact same Toggle logic.

View solution

CHALLENGE 3

Build the withLoading HOC from this chapter and use it to wrap a simple component. Then write a useLoading-style custom hook achieving the same "show loading until ready" behavior, and compare the two approaches.

📄 View solution

CHAPTER 5 QUICK REFERENCE

- **Compound components** — a family of components sharing implicit state via internal Context (`Tabs.Tab`, `Tabs.Panel`)
- **Render props** — a prop (often `children`) that's a function, letting the caller control rendering while the component owns behavior
- **HOCs** — a function taking a component, returning a new wrapped component with added behavior
- Compound components remain the standard for multi-part UI widgets; render props occasionally still earn their place
- HOCs are largely legacy — a custom hook (Intermediate Ch 4) almost always replaces them in new code
- **Next chapter:** performance profiling — measuring before optimizing, with React's own DevTools profiler

CHAPTER 6 OF 7

Performance Profiling and Optimization

COURSE 3 · CH 6

Performance Profiling and Optimization

The tools for actually measuring before reaching for Chapter 7's useMemo/useCallback/memo

Intermediate Chapter 7 introduced `memo`, `useMemo`, and `useCallback` with a clear warning attached: measure first, don't optimize by guessing. This chapter is that measurement — the React DevTools Profiler, recognizing the common causes of unnecessary re-rendering on sight, and the handful of other tools (virtualization, production builds, bundle analysis) that matter once an app is large enough for performance to be a real, measured concern.

The React DevTools Profiler

The React DevTools browser extension adds a "Profiler" tab alongside the regular Elements/Components inspector. Recording a session — clicking the record button, performing some interaction (a click, typing, a re-render), then stopping the recording — produces a flame graph: one bar per component that rendered during that window, sized by render duration, colored by relative cost. Clicking any individual render shows exactly which props or state changed for that component, and a tooltip explaining *why* it rendered (a parent re-rendered, props changed, hooks changed) — turning "this feels slow" into a specific, named component and a specific, named cause.

"HIGHLIGHT UPDATES WHEN COMPONENTS RENDER"

A settings toggle in the Profiler tab outlines every component on the actual page with a colored flash the instant it re-renders — no recording needed at all. It's the fastest way to spot an unexpectedly large swath of the UI flashing on every keystroke or click, exactly the kind of "the whole page re-renders when only one small thing changed" problem Chapter 7's tools exist to fix.

Recognizing the Common Causes

Three patterns explain the overwhelming majority of unnecessary re-renders, all already covered with their fixes in earlier chapters — this chapter's job is recognizing them in a profiler's output, not re-teaching the fixes themselves:

- **Parent cascade** — every child re-renders by default whenever its parent does, regardless of whether that child's own props changed. Fixed with `memo` (Intermediate Ch 7).
- **Unstable references** — a new object/array/function created every render defeats `memo`'s comparison even when the actual content is identical. Fixed with `useMemo`/`useCallback` (Intermediate Ch 7).
- **Context value changes** — every consumer of a context re-renders on any change to its `value`, including changes from unrelated state in the same provider. Fixed by memoizing the context's `value` object (also Intermediate Ch 7).

Long Lists — Virtualization

```
// npm install react-window
import { FixedSizeList } from "react-window";

function BigList({ items }) {
  return (
    <FixedSizeList height={400} itemCount={items.length} itemSize={35}>
      {({ index, style }) => <div style={style}>{items[index]}</div>}
    </FixedSizeList>
  );
}
```

`memo`/`useMemo` help when re-renders happen too often; they don't help when a single render itself is expensive simply because it's creating thousands of DOM nodes at once, the way `.map()` over a 5,000-item array (Fundamentals Chapter 6) would. **Virtualization** (also called "windowing") renders only the rows currently visible in the scrollable viewport, recycling the same handful of DOM nodes as the user scrolls — `react-window` is the standard library for this, used here as a render-props-style component (Chapter 5) passing each visible row's `index` and positioning `style`.

Always Profile a Production Build

```
# development mode (npm run dev) is NOT representative of real performance
npm run build
npm run preview   # serves the actual production build locally
```

A Vite dev server runs extra checks, warnings, and unoptimized code specifically to make development easier (better error messages, instant updates) — at the direct cost of real performance accuracy. Profiling in dev mode regularly shows render times several times slower than the same code in a real production build; always build and profile the production output before drawing any real conclusion about whether something is actually slow for an end user.

Bundle Size — Finding What's Actually Large

Before deciding what to lazy-load (Chapter 3), it's worth knowing what's actually contributing to bundle size in the first place — a bundle visualizer (e.g. `vite-bundle-visualizer` or the official `rollup-plugin-visualizer`) generates a treemap of every dependency's contribution to the final bundle, often revealing one unexpectedly large library is responsible for most of the weight, which is exactly the kind of thing worth wrapping in `lazy()` rather than guessing at random.

STILL DON'T OPTIMIZE WITHOUT A NUMBER

Every tool in this chapter exists to produce an actual measurement — a flame graph, a render-duration number, a bundle size in kilobytes — before any code changes. "This component feels like it might be slow" is not a measurement; a Profiler recording showing it taking 40ms across 200 renders in a 10-second interaction is. Chapter 7's warning bears repeating here with the tools finally in hand to back it up.

Coding Challenges

CHALLENGE 1

Build a small app with an unrelated counter and an expensive child list (deliberately not yet memoized). Use the React DevTools Profiler (or the "highlight updates" toggle) to confirm the list re-renders when the counter changes, then apply memo/useMemo from Intermediate Chapter 7 and confirm via the same tool that it's fixed.

View solution

CHALLENGE 2

Render a list of 2,000+ items with a plain .map() and observe scroll/render performance. Then rebuild the same list using react-window's FixedSizeList and compare.

View solution

CHALLENGE 3

Run npm run build followed by npm run preview on any project from this course, and profile a typical interaction in both the dev server and the production preview, noting the difference in reported render times.

View solution

CHAPTER 6 QUICK REFERENCE

- **React DevTools Profiler** — records renders into a flame graph, with a "why did this render" explanation per component
- **"Highlight updates"** — visually flashes re-rendering components on the live page, no recording needed
- Three common causes: **parent cascade**, **unstable references**, **context value changes** — all fixed in Intermediate Ch 7
- **Virtualization (react-window)** — renders only visible rows of a very long list
- Always profile a **production build** (`npm run build && npm run preview`) — dev mode numbers are misleading
- A **bundle visualizer** shows what's actually large, informing what's worth code-splitting (Ch 3)
- **Next chapter:** intro to Next.js / SSR — wrapping up the course with the framework first previewed in Project 7

CHAPTER 7 OF 7

Intro to Next.js and Server-Side Rendering

COURSE 3 · CH 7

Intro to Next.js and Server-Side Rendering

The final chapter — what Next.js is actually for, beyond the file-based routing used in Project 7

Project 7 used Next.js purely for its file-based routing, with everything else identical to a regular Vite app. Every project across all four courses has used the same rendering approach without naming it: **client-side rendering (CSR)** — the browser downloads a near-empty HTML file, then downloads and runs JavaScript that builds the entire page from scratch. This final chapter explains what Next.js adds on top of that, and why it matters.

The Limits of Client-Side Rendering

A pure CSR app's initial HTML is essentially just `<div id="root"></div>` — nothing meaningful exists on the page until the JavaScript bundle finishes downloading, parsing, and running React's first render. On a slow connection or an underpowered device, that's a real, visible delay before anything appears at all. It also means anything that reads raw HTML without running JavaScript — some search engine crawlers, social media link previews, accessibility tools — sees essentially nothing.

Server-Side Rendering and Hydration

SSR runs the React rendering logic on the server, for each incoming request, producing real, populated HTML sent straight to the browser — a user sees actual content immediately, before any JavaScript has even started downloading. The browser then downloads React's JavaScript in the background and **hydrates** that already-visible HTML — attaching event listeners and making it properly interactive — rather than re-building it from nothing.

Server Components vs Client Components

```
// app/page.js — a Server Component by default in Next.js's App Router
async function HomePage() {
  const posts = await fetch("https://api.example.com/posts").then((r) => r.json());

  return (
    <ul>
      {posts.map((post) => <li key={post.id}>{post.title}</li>)}
    </ul>
  );
}

export default HomePage;
```

In Next.js's App Router, every component is a **Server Component** by default — it runs only on the server, and crucially, none of its code is ever shipped to the browser at all, not even as part of the JavaScript bundle. Notice `HomePage` is simply an `async` function reading data with plain `await fetch(...)` — no `useState`, no `useEffect`, none of Intermediate Chapter 6's loading-state machinery, because there's no client-side loading state to manage: the data is already there by the time the HTML reaches the browser.

```
// app/Counter.js
"use client";

import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);
  return <button onClick={() => setCount(count + 1)}>{count}</button>;
}

export default Counter;
```

Anything needing `useState`, `useEffect`, event handlers, or any other browser-only behavior must be marked a **Client Component** with the `"use client"` directive at the very top of the file — this opts that specific component (and anything it imports) back into the familiar CSR-style behavior covered throughout the rest of this entire course, hydrated and interactive in the browser. A typical Next.js page mixes both: Server Components for the bulk of the static, data-heavy layout, with small Client Components dropped in exactly where real interactivity is needed.

FORGETTING "USE CLIENT" IS THE SINGLE MOST COMMON NEXT.JS ERROR

Calling `useState` (or any other hook, or attaching an `onClick`) inside a component without `"use client"` fails immediately with an error, since Server Components have no browser runtime to manage that state in at all. The fix is always the same: add `"use client"` as the very first line of that file. A second, subtler issue — a **hydration mismatch** — happens when a Server Component's rendered HTML and the client's first render disagree (e.g. using `Date.now()` or checking `window` directly inside a Server Component, which doesn't exist during that initial render);

React detects the mismatch and re-renders to "fix" it, but the visible flicker and console warning are a sign something rendered differently than expected between server and client.

STRATEGY	WHEN IT RENDERS	BEST FOR
`CSR (every project so far)`	Entirely in the browser, after JS loads	Highly interactive apps; SEO/initial load less critical
`SSR`	On the server, fresh, per request	Pages whose content changes per visit/user, needing fast first paint + SEO
`SSG (Static Generation)`	On the server, once, at build time	Content that's the same for everyone (marketing pages, blog posts)

Coding Challenges

CHALLENGE 1

Build a Next.js Server Component page that fetches a list from any free public API directly with async/await (no useState/useEffect at all), rendering the results.

View solution

CHALLENGE 2

Build a Counter Client Component (marked with "use client", using useState) and drop it into an otherwise server-rendered page alongside server-fetched content, confirming both the static content and the interactive counter work correctly on the same page.

View solution

CHALLENGE 3

Deliberately write a component using useState without the "use client" directive, run the app, and read the resulting error message carefully. Then add "use client" and confirm it resolves.

View solution

CHAPTER 7 QUICK REFERENCE

- **CSR** — everything used throughout this course; renders entirely in the browser after JS loads
- **SSR** — server renders real HTML per request; the browser then **hydrates** it into an interactive app

- **Server Components** (Next.js App Router default) — run only on the server; their code never reaches the browser
- **"use client"** — required for any component using hooks, event handlers, or other browser-only behavior
- Server Components can `await fetch(...)` directly — no loading state needed, since data arrives before the HTML does
- A **hydration mismatch** happens when server and client render disagree — watch for browser-only APIs used inside Server Components

🎉 That completes React Advanced — and with it, the entire React series: Fundamentals, Intermediate, Projects, and Advanced, 31 chapters and 7 project briefs in total.