React Intermediate

A Complete 7-Chapter Course

Topics covered:

useRef · Context API · useReducer · Custom Hooks

React Router · Fetching Data · Performance Basics

Exercises: 21 hands-on challenges with sample solutions

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

Table of Contents

CHAPTER 1 OF 7

useRef

COURSE 2 · CH 1

useRef

A place to keep a value that doesn't need to trigger a re-render — including direct access to a real DOM node

Welcome to Intermediate. Every piece of changing data covered so far has gone through `useState`, specifically because changing it needed to update the screen. Not everything fits that description — sometimes a value just needs to persist between renders without ever causing one, or a component needs to reach directly into the actual DOM node React rendered. `useRef` is the hook for both of those cases.

Accessing a Real DOM Node

```
import { useRef } from "react";

function SearchBox() {
  const inputRef = useRef(null);

  function handleFocusClick() {
    inputRef.current.focus();
  }

  return (
    <div>
      <input ref={inputRef} />
      <button onClick={handleFocusClick}>Focus the input</button>
    </div>
  );
}
```

`useRef(null)` creates a ref object with a single property, `current`, starting at `null`. Passing it to an element's special `ref` attribute tells React to set `inputRef.current` to that exact DOM node once it's rendered — from then on, `inputRef.current` *is* the real `<input>` element, with every normal DOM method (`.focus()`, `.scrollIntoView()`, reading `.offsetHeight`, and so on) available on it directly.

The Key Difference from useState

```
function RenderCounter() {
  const renderCount = useRef(0);
  renderCount.current = renderCount.current + 1;

  return <p>This component has rendered {renderCount.current} times.</p>;
}
```

Updating `renderCount.current` directly — no setter function, just a plain assignment — works perfectly well, and the new value *is* there on the next render. What it doesn't do is **cause** a re-render. If nothing else in this component ever updates, the displayed count would only ever change because something else (a parent re-rendering it, for instance) triggers a new render for a different reason. This is the core distinction from `useState`: a ref update is invisible to React's rendering system entirely.

	USESTATE	USEREF
Changing the value...	Triggers a re-render	Does not trigger a re-render
Read/write via...	`value` / setter function	`ref.current`, direct assignment
Good for...	Anything shown in the UI	DOM access, or data that doesn't affect what's rendered

Storing a Mutable Value Across Renders

```
function Stopwatch() {
  const [seconds, setSeconds] = useState(0);
  const intervalRef = useRef(null);

  function handleStart() {
    intervalRef.current = setInterval(() => setSeconds((s) => s + 1), 1000);
  }

  function handleStop() {
    clearInterval(intervalRef.current);
  }

  return (
    <div>
      <p>{seconds}s</p>
      <button onClick={handleStart}>Start</button>
      <button onClick={handleStop}>Stop</button>
    </div>
  );
}
```

Fundamentals Chapter 8's clock example stored its interval id as a plain variable inside the effect, cleaned up automatically when the effect re-ran. Here, the interval needs to be started and stopped from two *separate*

event handlers — `handleStart` and `handleStop` — rather than a single effect's cleanup, so the id needs to survive between renders without living inside any one function call. A ref is exactly built for that: a box that keeps holding its value across every re-render, completely independent of the render cycle itself.

DON'T READ OR WRITE A REF DURING RENDERING FOR VALUES THAT AFFECT THE UI

Reading `someRef.current` directly inside the JSX a component returns (rather than inside an event handler or effect) works technically, but breaks the assumption that rendering the same props/state twice produces the same output — since a ref's value can silently differ between two renders without React ever knowing to re-render in response. If a value needs to show up in the UI, that's the signal it belongs in `useState` instead, not a ref.

Coding Challenges

CHALLENGE 1

Build a component with a text input that automatically receives focus the moment the component first appears, using useRef together with useEffect (empty dependency array).

View solution

CHALLENGE 2

Build a component with a count state value and a button that increments it. Alongside it, track how many times the button has been clicked using a separate ref, and display both numbers — they should always match, but only one of them is responsible for the re-render that shows the updated count.

View solution

CHALLENGE 3

Build a Stopwatch component with Start and Stop buttons, storing the interval id in a ref so handleStart and handleStop can both access the same id across separate function calls.

View solution

CHAPTER 1 QUICK REFERENCE

- **useRef(initialValue)** — returns `{ current: initialValue }`, persisting across renders
- Changing `ref.current` does **not** trigger a re-render — unlike `useState`'s setter
- The **ref** attribute on a JSX element gives `ref.current` direct access to that real DOM node

- Good ref use cases: focusing an input, storing a timer/interval id across separate handlers, anything that doesn't need to appear in the UI
- If a value needs to be shown on screen, it belongs in **state**, not a ref
- **Next chapter:** the Context API — sharing data without prop drilling, first previewed back in Project 6

CHAPTER 2 OF 7

Context API and useContext

COURSE 2 · CH 2

Context API and useContext

Sharing a value with any component nested below, with no props passed through the layers in between

The shopping cart project briefly used Context to solve a real problem: a header badge, a product grid, and a cart page all needing the same data despite sitting nowhere near each other in the component tree. This chapter covers that mechanism properly — Context as React's built-in answer to the prop-drilling problem first raised back in Fundamentals Chapter 10.

The Three Pieces

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

// 1. Create the context
const ThemeContext = createContext(null);

// 2. Provide a value, somewhere up the tree
function App() {
  const [theme, setTheme] = useState("light");

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

// 3. Read it, anywhere below — no matter how deep
function ThemeButton() {
  const { theme, setTheme } = useContext(ThemeContext);
  return (
    <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>
      Current theme: {theme}
    </button>
  );
}
```

`createContext(null)` creates the context object itself, separately from any actual data — the `null` is just a fallback default. `<ThemeContext.Provider value={...}>` makes that `value` available to everything nested inside it; `ThemeButton` can sit any number of components below `App` — even with several unrelated components in between — and still reach the same `theme` / `setTheme` directly via `useContext(ThemeContext)`, without a single one of those in-between components ever mentioning theme at all.

Providing State and Functions Together

The shopping cart project's pattern — bundling state and the functions that update it into one object passed to `value` — is the standard shape for a real context, not just a single raw value. It's common enough to extract the whole thing into its own dedicated "provider" component, exactly as `CartProvider` did:

```
function ThemeProvider({ children }) {
  const [theme, setTheme] = useState("light");

  function toggleTheme() {
    setTheme((t) => (t === "light" ? "dark" : "light"));
  }

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}
```

`ThemeProvider` wraps `{children}` (the composition pattern from Fundamentals Chapter 9) rather than rendering anything visible itself — its entire job is owning the state and exposing it, leaving every consumer free to use it however it wants.

A Custom Hook for Cleaner Consumption

```
function useTheme() {
  return useContext(ThemeContext);
}

// usage — consumers never need to import ThemeContext directly
const { theme, toggleTheme } = useTheme();
```

Wrapping `useContext(ThemeContext)` in a small custom hook is an extremely common pattern in real codebases — it means every consumer imports one clearly-named function (`useTheme`) instead of needing to

know the underlying context object exists at all. Custom hooks get their own full treatment next chapter; this is a first glimpse of how naturally they pair with Context specifically.

CONTEXT ISN'T FREE, AND IT ISN'T FOR EVERYTHING

Every component consuming a context re-renders whenever that context's `value` changes — for state that changes rarely (theme, the logged-in user, locale), that's entirely fine. For state that changes very frequently (every keystroke, every animation frame), Context can cause far more re-rendering than passing props directly to just the few components that need it. When in doubt: reach for lifting state up first, and only bring in Context once prop drilling genuinely becomes painful, exactly as Fundamentals Chapter 10 first suggested.

SITUATION	REACH FOR...
`A few nearby components share state`	Lifting state up (props)
`Many distant components need the same rarely-changing value`	Context
`State changes very frequently and widely consumed`	A dedicated state library (Advanced Ch 1)

Coding Challenges

CHALLENGE 1

Build a ThemeContext with a ThemeProvider holding theme ("light"/"dark") and a toggleTheme function. Consume it in at least two unrelated components (e.g. a Header and a Footer) so both reflect the same theme and both have a way to toggle it.

View solution

CHALLENGE 2

Build an AuthContext storing a user (null when logged out, an object like { name } when logged in) plus login and logout functions. Consume it in a Header (showing "Welcome, [name]" or a login button) and a separate Dashboard component (showing content only when a user is logged in).

View solution

CHALLENGE 3

Take your ThemeContext from Challenge 1 and write a useTheme custom hook wrapping useContext(ThemeContext). Refactor every consumer to call useTheme() instead of useContext(ThemeContext) directly.

View solution

CHAPTER 2 QUICK REFERENCE

- **createContext(default)** — creates the context object itself
- **<Context.Provider value={...}>** — makes a value available to every nested consumer
- **useContext(Context)** — reads that value, from any depth, with no props passed through
- Bundle state + its updater functions into one `value` object — the standard shape for a real context
- A small custom hook (`useTheme`) wrapping `useContext` is a common, cleaner consumption pattern
- Context re-renders every consumer on change — best for rarely-changing, widely-needed values, not high-frequency state
- **Next chapter:** useReducer — managing more complex state transitions with a single dispatch function

CHAPTER 3 OF 7

useReducer

COURSE 2 · CH 3

useReducer

Centralizing every way a piece of state can change into a single, predictable function

The Kanban board and shopping cart projects both had state that could change in several distinct ways — add, remove, update quantity, move between columns — each handled by its own separate function calling `setItems` or `setColumns` with slightly different logic. As the number of "ways state can change" grows, that logic ends up scattered across many handlers. `useReducer` centralizes all of it into one place: a single function describing every possible state transition.

A Reducer Is Just a Function

```
function counterReducer(state, action) {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    case "reset":
      return { count: 0 };
    default:
      return state;
  }
}
```

A reducer takes the **current state** and an **action** describing what happened, and returns the **new state** — nothing more. It's a plain function, with no React-specific code inside it at all; every possible transition lives in one readable place instead of being spread across separate `handleIncrement` / `handleDecrement` / `handleReset` functions.

useReducer in a Component

```jsx
import { useReducer } from "react";

function Counter() {
  const [state, dispatch] = useReducer(counterReducer, { count: 0 });

  return (
    <div>
      <p>{state.count}</p>
      <button onClick={() => dispatch({ type: "increment" })}>+1</button>
      <button onClick={() => dispatch({ type: "decrement" })}>-1</button>
      <button onClick={() => dispatch({ type: "reset" })}>Reset</button>
    </div>
  );
}
```

`useReducer(reducer, initialState)` returns `[state, dispatch]` — `state` is read exactly like `useState`'s value, but instead of a setter, `dispatch` sends an **action object** to the reducer, which decides what the new state should be. No button handler needs to know *how* to update the count — each one just describes *what happened* ("increment"), leaving the actual logic entirely inside the reducer.

Actions with a Payload

```jsx
function todosReducer(state, action) {
  switch (action.type) {
    case "add":
      return [...state, { id: action.payload.id, text: action.payload.text, done: false }];
    case "toggle":
      return state.map((todo) =>
        todo.id === action.payload.id ? { ...todo, done: !todo.done } : todo
      );
    case "remove":
      return state.filter((todo) => todo.id !== action.payload.id);
    default:
      return state;
  }
}

// dispatching with extra information attached:
dispatch({ type: "toggle", payload: { id: 3 } });
```

`payload` is just a convention — a place to attach whatever extra information a particular action needs (which todo's `id` to toggle or remove, the text for a new one). Project 1's todo list scattered this same add/toggle/remove logic across three separate state-updating functions; written as a reducer, all three transitions live together, each one still using the same immutable-update rules (spread, `.map()`, `.filter()`) from Fundamentals Chapter 3 and Project 1 — just consolidated into a single function instead of three.

Pairing useReducer with Context

`useReducer` and Context combine naturally for global state: the reducer's `state` becomes the context's `value`, and `dispatch` itself can be exposed through context too, letting any deeply nested component dispatch an action without ever needing the state-updating logic passed down to it as props. This combination is genuinely how some real apps implement global state without reaching for an external library at all — a more powerful version of the `CartProvider` pattern from Project 6, with one centralized reducer instead of several separate updater functions.

A REDUCER MUST STAY PURE

A reducer should never mutate its `state` argument directly, call an API, set a timer, or do anything else with a side effect — given the same `state` and `action`, it must always return the same result, with no exceptions. It's also easy to forget the `default` case in the `switch` — without one, dispatching an unrecognized action type returns `undefined` instead of the unchanged state, silently wiping out everything.

SITUATION	REACH FOR...
One or two simple, independent values	useState
Several related values that change together, in many distinct ways	useReducer
The same reducer state needed deep in the tree, widely	useReducer + Context

Coding Challenges

CHALLENGE 1

Build a Counter component using useReducer with "increment", "decrement", and "reset" action types, matching the chapter's example exactly.

View solution

CHALLENGE 2

Refactor a todo list (add, toggle complete, remove) to use a single useReducer instead of separate useState-driven handlers, using "add"/"toggle"/"remove" action types with a payload carrying the relevant id/text.

View solution

CHALLENGE 3

Build a NotificationContext using useReducer + Context together, supporting "show" (adds a message) and "dismiss" (removes one by id) actions, consumed by two unrelated components — one that triggers a notification, one that displays the current list of them.

View solution

CHAPTER 3 QUICK REFERENCE

- A **reducer** is a pure function: `(state, action) => newState`
- **useReducer(reducer, initialState)** returns `[state, dispatch]`
- **dispatch({ type, payload })** — describes what happened, not how to update state
- Centralizes many related state transitions in one place, instead of scattering them across handlers
- Pairs naturally with **Context** for global state, without an external library
- A reducer must stay **pure** — no mutation, no side effects, and always include a `default` case
- **Next chapter:** custom hooks — extracting reusable logic of your own, the way useTheme did in Chapter 2

CHAPTER 4 OF 7

Custom Hooks

COURSE 2 · CH 4

Custom Hooks

Extracting reusable stateful logic into a function of your own — exactly what useTheme already did, generalized

Chapter 2's `useTheme` was already a custom hook, introduced without dwelling on the concept itself. A custom hook is nothing more than a regular JavaScript function whose name starts with `use`, which calls other hooks inside it — there's no special syntax, no registration step, no library involved. It exists purely to pull repeated *stateful* logic (state plus the behavior around it) out of components and into something reusable, the same way a regular function extracts repeated non-stateful logic.

The Simplest Possible Custom Hook

```
function useToggle(initialValue = false) {
  const [value, setValue] = useState(initialValue);

  function toggle() {
    setValue((v) => !v);
  }

  return [value, toggle];
}

// usage — looks just like useState, but with toggle behavior built in
function Switch() {
  const [isOn, toggleIsOn] = useToggle();
  return <button onClick={toggleIsOn}>{isOn ? "On" : "Off"}</button>;
}
```

`useToggle` wraps a `useState` call and the boolean-flip logic that goes with it (the same pattern as Fundamentals Chapter 3's `LightSwitch`), so any component needing toggle behavior can call one hook instead of writing the flip function itself every time. It returns a tuple, like `useState` does, purely as a convention — a custom hook can return anything at all (an object, a single value, several values).

useLocalStorage — Syncing State with the Browser

```
function useLocalStorage(key, initialValue) {
  const [value, setValue] = useState(() => {
    const saved = localStorage.getItem(key);
    return saved ? JSON.parse(saved) : initialValue;
  });

  useEffect(() => {
    localStorage.setItem(key, JSON.stringify(value));
  }, [key, value]);

  return [value, setValue];
}

// usage — drop-in replacement for useState, now persisted automatically
const [notes, setNotes] = useLocalStorage("notes", []);
```

This is exactly the load/save pair Project 3's stretch goals suggested extracting — two effects' worth of `localStorage` logic, now living in one hook instead of being duplicated in every component that needs persistence. Passing a **function** to `useState` (rather than a plain value) is worth noting here: that initializer function only runs once, on the very first render, rather than re-reading `localStorage` on every single re-render.

useDebounce — Delaying a Fast-Changing Value

```
function useDebounce(value, delay) {
  const [debouncedValue, setDebouncedValue] = useState(value);

  useEffect(() => {
    const timeoutId = setTimeout(() => setDebouncedValue(value), delay);
    return () => clearTimeout(timeoutId);
  }, [value, delay]);

  return debouncedValue;
}

// usage
const [query, setQuery] = useState("");
const debouncedQuery = useDebounce(query, 400);
// fetch only when debouncedQuery changes, not on every keystroke in query
```

This is the Book Search project's debounce logic, lifted out of a component and into a reusable hook — exactly the stretch goal it suggested. `useDebounce` takes a fast-changing value (`query`, updated on every keystroke) and returns a second value that only catches up after `delay` milliseconds of no further changes, using the same `setTimeout` + cleanup pattern from Fundamentals Chapter 8 and Project 4, just packaged for reuse anywhere a debounced value is needed.

WHEN SOMETHING DESERVES TO BECOME A CUSTOM HOOK

Not every reusable bit of logic needs to be a hook — a plain helper function is the right tool when no state or other hook is involved at all. The signal for a *custom hook* specifically is reusable logic that depends on `useState`, `useEffect`, `useContext`, or another hook internally. If two or more components are duplicating the same stateful behavior, that's usually the moment to extract it.

CUSTOM HOOKS STILL FOLLOW THE RULES OF HOOKS

Because a custom hook calls other hooks internally, it inherits the exact same rules from Fundamentals Chapter 3: call it only at the top level of a component (or another hook) — never inside a condition, loop, or nested function. The `use` prefix isn't just a naming convention; tools like the ESLint hooks plugin specifically look for that prefix to know which functions to apply these rules to.

Coding Challenges

CHALLENGE 1

Write the useToggle hook from this chapter and use it in two different components — one toggling a sidebar's open/closed state, one toggling a "show password" boolean on a password input.

View solution

CHALLENGE 2

Write the useLocalStorage hook from this chapter, and use it to persist a simple counter's current value, confirming it survives a page refresh.

View solution

CHALLENGE 3

Write the useDebounce hook from this chapter. Build a search input where the raw value updates on every keystroke, and a separate debounced value (400ms) is displayed below it once typing pauses — enough to see the two values catch up to each other after a short delay.

View solution

CHAPTER 4 QUICK REFERENCE

- A custom hook is just a function starting with **use** that calls other hooks inside it

- Extract one when several components duplicate the same **stateful** logic — not just any repeated code
- **useToggle** — boolean state + a flip function, wrapped for reuse
- **useLocalStorage** — a drop-in `useState` replacement that persists automatically
- **useDebounce** — returns a value that only catches up after a pause in changes
- Custom hooks still follow the **rules of hooks** — top-level calls only, no conditionals
- **Next chapter:** React Router — multiple pages in a single-page app, first previewed in Project 4

CHAPTER 5 OF 7

React Router

COURSE 2 · CH 5

React Router

Multiple "pages" inside a single-page app, navigated without ever reloading the browser

Project 4 used four pieces of React Router with a quick primer to get a search page and a detail page working. This chapter covers routing properly — including the two pieces that primer left out entirely: programmatic navigation, and sharing a layout (like a persistent sidebar or header) across several routes.

The Setup, Recapped

```
// npm install react-router-dom

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

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/about" element={<AboutPage />} />
      </Routes>
    </BrowserRouter>
  );
}
```

`BrowserRouter` wraps the whole app once, near the top. `Routes` looks through its `Route` children for the one whose `path` matches the current URL, and renders only that one's `element`.

Link vs NavLink

```jsx
import { Link, NavLink } from "react-router-dom";

<Link to="/about">About</Link>

<NavLink
  to="/about"
  className={({ isActive }) => (isActive ? "nav-link active" : "nav-link")}
>
  About
</NavLink>
```

Both navigate without a full page reload, exactly like Project 4's `Link`. `NavLink` is specifically for navigation menus — it knows whether its own `to` matches the current URL, and exposes that as `isActive` inside a function passed to `className` (or `style`), making "highlight the current page in the nav" straightforward without manually comparing the URL yourself.

useParams and useNavigate

```jsx
import { useParams, useNavigate } from "react-router-dom";

function ProductDetailPage() {
  const { id } = useParams();
  const navigate = useNavigate();

  function handleDelete() {
    // after some action completes, send the user somewhere else
    navigate("/products");
  }

  return (
    <div>
      <p>Showing product {id}</p>
      <button onClick={handleDelete}>Delete and go back</button>
    </div>
  );
}
```

`useParams()` was already used in Project 4 to read a dynamic URL segment (`/product/:id` → `{ id }`). `useNavigate()` is new here — it returns a function for navigating **programmatically**, from inside an event handler or effect, rather than from a `Link` the user clicks. This is the right tool for redirecting after a form submits, after a delete completes, or after a login succeeds.

Nested Routes and a Shared Layout

```jsx
import { Outlet } from "react-router-dom";

function DashboardLayout() {
  return (
    <div>
      <Sidebar />
      <main>
        <Outlet />   {/* the matched child route renders here */}
      </main>
    </div>
  );
}

// in the route definitions:
<Route path="/dashboard" element={<DashboardLayout />}>
  <Route path="profile" element={<ProfilePage />} />
  <Route path="settings" element={<SettingsPage />} />
</Route>
```

Nesting `Route` elements means the parent's `element` (`DashboardLayout`, with its sidebar) renders for *every* URL starting with `/dashboard`, while `<Outlet />` marks exactly where the matching child route's content should appear inside it. Visiting `/dashboard/profile` renders `DashboardLayout` with `ProfilePage` dropped into the `Outlet` — the sidebar never unmounts or re-renders just because the inner page changed, which is exactly the composition pattern (children/Outlet as a flexible slot) from Fundamentals Chapter 9, applied to routing.

A Catch-All "Not Found" Route

```jsx
<Route path="*" element={<NotFoundPage />} />
```

A `Route` with `path="*"`, placed last among the route definitions, matches any URL that didn't match an earlier, more specific route — the standard way to show a friendly 404 page instead of a blank screen for a mistyped or stale URL.

> **A PLAIN <A HREF> DEFEATS THE ENTIRE POINT**
>
> Using a regular `<a href="/about">` instead of `<Link to="/about">` triggers a real full-page browser navigation — reloading the entire app from scratch, losing any in-memory state (cart contents, form data, anything not in `localStorage`), exactly the cost React Router exists to avoid. `Link` / `NavLink` intercept the click and update the URL without a reload; a plain anchor tag never does.

Coding Challenges

CHALLENGE 1

Build a 3-page app (Home, About, Contact) with a nav bar using NavLink for each, styling the currently active link differently from the others.

View solution

CHALLENGE 2

Build a hardcoded list of products with Link to /product/:id for each, and a ProductDetailPage that reads the id via useParams and shows the matching product's details.

View solution

CHALLENGE 3

Build a layout route with a persistent sidebar (using Outlet) wrapping two nested child pages, plus a catch-all "*" route rendering a NotFoundPage for any unmatched URL.

View solution

CHAPTER 5 QUICK REFERENCE

- **BrowserRouter** wraps the app; **Routes/Route** map a path to an element
- **Link** — basic navigation; **NavLink** — adds active-state awareness for nav menus
- **useParams()** — reads dynamic URL segments; **useNavigate()** — navigates programmatically
- **Nested routes + <Outlet />** — a shared layout (sidebar, header) wrapping several child pages
- **path="*"**, placed last — catches any URL nothing else matched, for a 404 page
- Always use `Link` / `NavLink`, never a plain `<a href>`, for in-app navigation
- **Next chapter:** fetching data — proper loading/error patterns beyond the basics from Fundamentals Ch 8

CHAPTER 6 OF 7

Fetching Data

COURSE 2 · CH 6

Fetching Data

Doing the loading/error/race-condition handling properly, after using the basics informally since Fundamentals Chapter 8

Fundamentals Chapter 8 introduced fetching with a minimal example; the Weather App project formalized it into a single `status` value (`idle` / `loading` / `error` / `success`); the Book Search project flagged, but didn't fix, a real correctness problem — a slow, stale request finishing after a newer one and overwriting it with outdated data. This chapter closes that gap properly, and ends with the same fetch logic extracted into a reusable hook.

Async Inside useEffect — The Right Shape

```
useEffect(() => {
  async function loadData() {
    const response = await fetch(url);
    const data = await response.json();
    setData(data);
  }

  loadData();
}, [url]);
```

The function passed directly to `useEffect` can never be `async` itself — React expects that function to return either nothing or a cleanup function, and an `async` function always returns a Promise instead, which React would mistake for a cleanup function and try to call later. The fix is always the same: define a separate `async` function *inside* the effect, and call it immediately.

Proper Error Handling

```
useEffect(() => {
  async function loadData() {
    setStatus("loading");
    try {
      const response = await fetch(url);
      if (!response.ok) {
        throw new Error(`Request failed: ${response.status}`);
      }
      const data = await response.json();
      setData(data);
      setStatus("success");
    } catch (err) {
      setStatus("error");
    }
  }

  loadData();
}, [url]);
```

A crucial detail easy to miss: `fetch` only rejects (triggering `catch`) on a genuine network failure — a 404 or a 500 response is still a "successful" fetch as far as the Promise is concerned. `response.ok` is what actually distinguishes a real HTTP success (status 200–299) from an error status, which is why it needs to be checked and thrown manually to land in the same `catch` block as a true network error.

The Race Condition, Properly Fixed

```
useEffect(() => {
  let ignore = false;

  async function loadData() {
    setStatus("loading");
    const response = await fetch(url);
    const data = await response.json();
    if (!ignore) {
      setData(data);
      setStatus("success");
    }
  }

  loadData();
  return () => { ignore = true; };
}, [url]);
```

This is the fix Project 4 flagged but didn't implement. When `url` changes before the previous fetch finishes, React runs the **cleanup function** (from Fundamentals Chapter 8) for the old effect before starting the new one — setting that old effect's own `ignore` flag to `true`. When the old, now-stale response finally arrives,

`if (!ignore)` is `false`, and its outdated data is correctly discarded rather than overwriting the newer request's result.

> **AN ALTERNATIVE: ABORTCONTROLLER**
>
> `AbortController` can cancel the underlying network request itself, rather than just ignoring its result after the fact — slightly more efficient, since the browser stops downloading data nobody needs anymore. The ignore-flag approach above is simpler to read and reason about for most cases, and is a perfectly reasonable default; reach for `AbortController` specifically when actually cancelling the network request (not just its effect) matters.

Extracting useFetch

```
function useFetch(url) {
  const [data, setData] = useState(null);
  const [status, setStatus] = useState("idle");

  useEffect(() => {
    let ignore = false;
    setStatus("loading");

    async function loadData() {
      try {
        const response = await fetch(url);
        if (!response.ok) throw new Error("Request failed");
        const result = await response.json();
        if (!ignore) { setData(result); setStatus("success"); }
      } catch (err) {
        if (!ignore) setStatus("error");
      }
    }

    loadData();
    return () => { ignore = true; };
  }, [url]);

  return { data, status };
}

// usage — every component fetching data shrinks to this
const { data, status } = useFetch(`/api/products/${id}`);
```

Everything in this chapter — loading state, error handling, race-condition protection — now lives in exactly one place, exactly the custom-hook extraction principle from Chapter 4. Every component that needs to fetch something calls `useFetch(url)` and gets correct behavior automatically, instead of re-implementing this same logic (and re-introducing the same bugs) in every single component that fetches data.

THIS ENTIRE CHAPTER, SOLVED FOR YOU

Project 7's Dashboard capstone used **React Query**, covered fully in Advanced Chapter 2 — its `useQuery` hook handles loading state, errors, race conditions, and caching, all without writing any of the code above by hand. Understanding what's actually happening underneath (this chapter) makes React Query's behavior feel like a natural extension rather than unexplained magic once it's introduced properly.

Coding Challenges

CHALLENGE 1

Build a component that fetches a single resource (any free public API) using the async-function-inside-useEffect pattern, with full try/catch error handling and a status value covering loading/error/success.

View solution

CHALLENGE 2

Build a component that fetches data based on an id prop, with two buttons that rapidly switch between two different ids. Add the ignore-flag cleanup pattern, and confirm (e.g. via an artificial delay) that switching ids quickly never shows the wrong id's stale data.

View solution

CHALLENGE 3

Extract the chapter's useFetch hook and use it in two different components fetching two different URLs, confirming both work independently with their own data/status.

View solution

CHAPTER 6 QUICK REFERENCE

- Never make the `useEffect` callback itself `async` — define an inner `async` function and call it
- **response.ok** must be checked manually — `fetch` doesn't reject on a 404/500 by itself
- The **ignore-flag pattern** (or `AbortController`) prevents a stale, slow response from overwriting newer data
- **useFetch(url)** — extracting all of the above into one reusable custom hook
- React Query (Advanced Ch 2) automates everything covered in this chapter

- **Next chapter:** performance basics — useMemo, useCallback, React.memo

CHAPTER 7 OF 7

Performance Basics — useMemo, useCallback, React.memo

COURSE 2 · CH 7

Performance Basics — useMemo, useCallback, React.memo

Skipping unnecessary re-renders and recalculations, and directly fixing the Context re-render warning from Chapter 2

Chapter 2 warned that every consumer of a context re-renders whenever its `value` changes — and left that as a tradeoff to accept for rarely-changing state. This chapter gives the actual tools for reducing unnecessary re-rendering generally, closing with the fix for that exact warning.

React.memo — Skipping a Component's Re-render

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

const ExpensiveList = memo(function ExpensiveList({ items }) {
  console.log("ExpensiveList rendered");
  return <ul>{items.map((item) => <li key={item}>{item}</li>)}</ul>;
});

function App() {
  const [count, setCount] = useState(0);
  const items = ["Apple", "Banana", "Cherry"];

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>
      <ExpensiveList items={items} />
    </div>
  );
}
```

Without `memo`, clicking the button re-renders `App` — and by default, every child re-renders right along with its parent, regardless of whether that child's own props actually changed. `memo(ExpensiveList)` wraps the component so React first compares its new props against the previous render's props; if they're the same, React skips re-rendering it entirely, reusing the previous output instead.

MEMO ONLY HELPS IF THE PROPS GENUINELY DON'T CHANGE

The `items` array above is recreated as a brand-new array on every single render of `App` — even though its contents look identical, it's a *different* array in memory each time, and `memo`'s comparison is by reference, not by deep content. As written, clicking the button would still cause `ExpensiveList` to re-render, because `items` "changed" every time, defeating the whole point. This is exactly what `useMemo` exists to fix.

useMemo — Memoizing a Value

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

  const items = useMemo(() => ["Apple", "Banana", "Cherry"], []);

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>
      <ExpensiveList items={items} />
    </div>
  );
}
```

`useMemo(calculateValue, dependencies)` only re-runs `calculateValue` when something in `dependencies` has actually changed since the last render — with an empty array, the array of fruit strings is computed exactly once and the *same* array reference is reused on every subsequent render. Now `ExpensiveList`'s `memo` comparison sees the identical `items` reference each time `count` changes, and correctly skips re-rendering.

useCallback — Memoizing a Function

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

  const handleSelect = useCallback((item) => {
    console.log("selected:", item);
  }, []);

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>
      <ExpensiveList items={items} onSelect={handleSelect} />
    </div>
  );
}
```

A regular function defined inside a component (`onSelect={(item) => ...}` written inline, or even a named function declared in the component body) is also a *new value* every render — functions are compared by reference too, the same as arrays and objects. `useCallback(fn, dependencies)` is `useMemo`'s counterpart specifically for functions: it returns the exact same function reference across renders as long as the dependencies haven't changed, which is what lets a memoized child receiving that function as a prop actually skip re-rendering.

Fixing the Context Re-render Warning from Chapter 2

```
function ThemeProvider({ children }) {
  const [theme, setTheme] = useState("light");

  const toggleTheme = useCallback(() => {
    setTheme((t) => (t === "light" ? "dark" : "light"));
  }, []);

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

  return <ThemeContext.Provider value={value}>{children}</ThemeContext.Provider>;
}
```

Chapter 2's `{ theme, toggleTheme }` object literal, written directly inside `value={{ ... }}`, was a brand-new object every time `ThemeProvider` rendered for *any* reason — even one completely unrelated to theme — forcing every consumer to re-render along with it. Wrapping it in `useMemo` means `value` only becomes a genuinely new object when `theme` itself actually changes, exactly addressing the warning raised back then.

> **DON'T REACH FOR THESE BY DEFAULT — MEASURE FIRST**
>
> `useMemo` and `useCallback` have their own (small) cost, and wrapping every single value and function in them "just in case" adds real complexity without necessarily improving anything — most components re-render fast enough that it's genuinely not worth thinking about. These tools earn their place specifically around expensive computations, large lists, and components wrapped in `memo` that need a stable prop identity to actually benefit. When performance hasn't been measured as an actual problem, plain `useState`/regular functions are usually the better default.

TOOL	MEMOIZES...	USE IT FOR...
`React.memo(Component)`	A component's render output	Skipping a child's re-render when its props are unchanged
`useMemo(fn, deps)`	A computed value	Expensive calculations, or stable object/array identity for memo'd children
`useCallback(fn, deps)`	A function reference	Stable function identity for memo'd children's props

Coding Challenges

CHALLENGE 1

Build a parent with an unrelated counter and a list component wrapped in React.memo, logging to the console every time the list renders. Use useMemo to keep the list's items array stable, and confirm in the console that incrementing the counter no longer re-renders the list.

View solution

CHALLENGE 2

Take Challenge 1's setup and add an onSelect handler prop passed to the memoized list. First confirm (via the console log) that an inline arrow function still causes the list to re-render despite memo. Then fix it with useCallback.

View solution

CHALLENGE 3

Take your ThemeContext/ThemeProvider from Chapter 2, add an unrelated piece of state to ThemeProvider (e.g. a render counter, logged whenever any consumer renders), and apply useMemo to the context's value object to confirm consumers stop re-rendering when that unrelated state changes.

View solution

CHAPTER 7 QUICK REFERENCE

- **React.memo(Component)** — skips re-rendering a component when its props are unchanged (by reference)
- **useMemo(fn, deps)** — recomputes a value only when its dependencies change; keeps object/array identity stable
- **useCallback(fn, deps)** — keeps a function's identity stable across renders
- Objects, arrays, and functions are compared **by reference** — a new one each render defeats `memo` even with identical contents
- Wrap a context's `value` in **useMemo** to stop unrelated state changes from re-rendering every consumer
- Measure before optimizing — these tools have real value, but aren't free, and most components don't need them
- **That's React Intermediate complete** — Advanced picks up with state management beyond Context next