React Projects

7 Project Briefs — Beginner to Capstone

Projects covered:

Todo List · Weather App · Notes App (localStorage) · Book Search (Router + debounce)
Kanban Board (drag-and-drop) · Shopping Cart (Context) · Dashboard Capstone (React Query + Next.js)

Format: Requirements, component breakdown, build order, and stretch goals — no single solution

Style: A4 · Dark-theme code examples · Open-ended, project-brief format

Table of Contents

PROJECT 1 OF 7

Todo List

PROJECT 1

Todo List

The classic first project — add, complete, and remove items from a list, all driven by state

DIFFICULTY: BEGINNER

BUILDS ON: FUNDAMENTALS CH 1–7

Unlike the chapter challenges, this is a project brief, not a graded exercise — there's no single correct solution file. Instead, you get requirements, a suggested structure, and stretch goals; how you actually build it is up to you. A todo list might seem simple, but it touches almost every Fundamentals concept at once: state, lists, conditional rendering, controlled inputs, and component composition, all working together in one small app.

SKILLS THIS PROJECT EXERCISES

useState · Controlled inputs · .map() + key · Conditional rendering · Lifting state up · Props & callbacks

Requirements

- A text input and "Add" button for entering a new todo — pressing Enter should also work, not just clicking the button.
- Each todo displays its text and has a way to mark it complete (e.g. a checkbox) and a way to delete it.
- Completed todos should look visually distinct from incomplete ones (e.g. strikethrough text, dimmed color).
- The input should clear itself after a todo is successfully added.
- Adding an empty/blank todo should be prevented.
- If the list is empty, show a friendly message (e.g. "No todos yet — add one above!") instead of an empty list.

Suggested Component Breakdown

```
App
└── TodoApp                  // holds the array of todos in state
    ├── TodoForm             // controlled input + add button
    └── TodoList             // maps over todos, renders one TodoItem per todo
        └── TodoItem (×N)    // checkbox, text, delete button — one per todo
```

This isn't the only valid structure — it's a starting point. The key decision worth sticking to: the array of todos itself should live in `TodoApp` (the lowest common ancestor of the form and the list), with `TodoForm` and `TodoItem` each receiving only the props and callbacks they specifically need, exactly as covered in Fundamentals Chapter 10.

A Reasonable Build Order

1. Get a static version working first — hardcode a small array of 2–3 todo objects (`{ id, text, completed }`) and render them with `TodoList`/`TodoItem` before wiring up any state changes.
2. Move that array into `useState` in `TodoApp`, so it's now real state instead of a hardcoded constant.
3. Wire up `TodoForm` to add a new todo object to the array on submit, generating a unique `id` for each one (e.g. `Date.now()` is a quick option for a small project like this).
4. Add the "toggle complete" behavior — clicking a todo's checkbox should flip its `completed` value without affecting any other todo in the array.
5. Add the delete behavior — removing exactly one todo from the array by its `id`, leaving the rest unchanged.
6. Add the empty-state message and the blank-input guard last, once the core add/toggle/delete flow is solid.

STRETCH GOALS

- Add an "edit" mode — double-clicking a todo's text turns it into an editable input.
- Add filter buttons (All / Active / Completed) above the list.
- Show a live count of remaining (incomplete) todos.

- Persist the list to `localStorage` so it survives a page refresh (a preview of Intermediate Chapter 6's data-handling patterns).
- Add a "Clear completed" button that removes every completed todo at once.

This project has no single solution file — the requirements above and your own judgment are the spec. Compare notes against the suggested component breakdown once you have something working, not before.

PROJECT 2 OF 7

Weather App

PROJECT 2

Weather App

Search a city, fetch real data, and handle loading and error states properly

DIFFICULTY: BEGINNER / INTERMEDIATE

BUILDS ON: FUNDAMENTALS CH 1–8

Project 1 worked entirely with local state — nothing ever left the browser. This project introduces a real external dependency: a weather API. That single change brings a whole new category of things to handle correctly — the data isn't there immediately, the request can fail, and the UI needs to make sense in all three states (loading, error, and loaded), not just the happy path.

SKILLS THIS PROJECT EXERCISES

useEffect / fetch | Loading & error state | Controlled search input | Conditional rendering | async/await

A FREE, NO-API-KEY WEATHER SOURCE

Open-Meteo provides a free weather API that doesn't require signing up for an API key, which makes it a good fit for a learning project. It needs latitude/longitude rather than a city name directly — Open-Meteo's own free geocoding endpoint converts a typed city name into coordinates first, so the app actually makes two fetches per search: one to resolve the city name, one to get the weather for those coordinates.

Requirements

- A text input where the user types a city name, with a search button (or Enter-to-search).
- While a search is in progress, show a clear loading indicator instead of stale or blank content.
- If the city can't be found, or the request fails, show a readable error message — not a blank screen or a crash.
- On success, display at minimum: the city name, current temperature, and a short description (e.g. "Clear sky," "Light rain").
- Searching for a new city should correctly replace the previously shown result, not append to it.

- An empty search input should not trigger a request at all.

Suggested Component Breakdown

```
App
└── WeatherApp          // holds city, weatherData, status state (idle/loading/error/success)
    ├── SearchForm      // controlled input + search button
    └── WeatherDisplay  // reads status and renders the matching UI for each case
```

The `status` state is the important design decision here — rather than separate `isLoading` / `hasError` booleans that could theoretically contradict each other, a single status value (`"idle" | "loading" | "error" | "success"`) guarantees the UI is always in exactly one well-defined state, using the lookup-object/conditional patterns from Fundamentals Chapter 5.

A Reasonable Build Order

1. Build `SearchForm` first with just a controlled input and a submit handler that logs the typed city name — no fetching yet.
2. Manually call the geocoding endpoint once in the browser/Postman/curl for a known city, to see the actual JSON shape you'll be working with before writing any fetch code.
3. Wire up the geocoding fetch inside an async function triggered on form submit, storing the resolved coordinates in state.
4. Add the second fetch for the actual weather data once coordinates are available, setting `status` to `"loading"` before it starts and `"success"` once it resolves.
5. Wrap both fetches in a try/catch, setting `status` to `"error"` (with a stored error message) if either one fails.
6. Build out `WeatherDisplay` last, rendering different JSX for each of the four status values.

STRETCH GOALS

- Show a multi-day forecast, not just current conditions.
- Add a Celsius/Fahrenheit toggle.
- Use the browser's Geolocation API to default to the user's current location on first load.
- Cache recent searches so re-searching the same city doesn't re-fetch immediately.
- Add simple weather-appropriate icons or background colors based on the description returned.

This project has no single solution file — the requirements above and your own judgment are the spec. Compare notes against the suggested component breakdown once you have something working, not before.

PROJECT 3 OF 7

Notes App with Local Storage Persistence

PROJECT 3

Notes App with Local Storage Persistence

Create, edit, and delete notes that survive a page refresh — no backend required

DIFFICULTY: INTERMEDIATE | BUILDS ON: FUNDAMENTALS CH 1–9

Project 1's todos vanished the moment the page reloaded — every piece of state was reset to nothing. This project adds **persistence**: saving the current data to the browser's `localStorage` on every change, and reading it back when the app first loads, so a note typed today is still there tomorrow. It's also the first project with a genuine "edit an existing item in place" flow, rather than just adding/removing whole items.

SKILLS THIS PROJECT EXERCISES

localStorage | useEffect (load + save) | JSON.stringify / parse | Editable list items | Controlled textarea | Component composition

Requirements

- A way to create a new note with at least a title and a body of text.
- All existing notes are listed, each showing its title and a preview (or full body) of its content.
- Clicking a note lets you edit its title/body in place, saving the changes back to the same note.
- Each note can be deleted individually.
- The entire notes list is saved to `localStorage` whenever it changes — adding, editing, or deleting a note.
- On page load/refresh, any previously saved notes are read back from `localStorage` and displayed — not lost.

Suggested Component Breakdown

```
App
└── NotesApp              // holds the notes array in state; loads/saves localStorage
    ├── NoteForm          // title + body inputs, "Add Note" button
    └── NoteList          // maps over notes, renders one NoteItem per note
        └── NoteItem (×N) // view mode + edit mode for one note, plus delete button
```

`NoteItem` is the trickiest piece — it needs its own small piece of local state (something like `isEditing`) to decide whether it's currently showing the note's static text or an editable form for it. That's a perfectly reasonable case for state living inside `NoteItem` itself rather than being lifted up, since no other component needs to know whether one particular note is currently being edited.

A Reasonable Build Order

1. Build the add/list/delete flow first, exactly like Project 1's todo list, with notes only living in memory (no `localStorage` yet).
2. Add a `useEffect` in `NotesApp` with the notes array as its dependency, calling `localStorage.setItem("notes", JSON.stringify(notes))` every time it changes.
3. Add a second `useEffect` (with an empty dependency array, running once on mount) that reads `localStorage.getItem("notes")`, parses it with `JSON.parse`, and sets it as the initial notes state — handling the case where nothing's been saved yet.
4. Refresh the page after adding a few notes to confirm persistence is actually working before building anything else.
5. Add edit mode to `NoteItem` last — its own `isEditing` state, a controlled form shown only while editing, and a "Save" action that updates the matching note in the parent's array.

LOCALSTORAGE ONLY STORES STRINGS

`localStorage.setItem` can't store a JavaScript array or object directly — it needs to be converted to a string first with `JSON.stringify`, and converted back with `JSON.parse` when reading it. Forgetting either step is one of the most common bugs in a project like this — usually showing up as `"[object Object]"` being stored literally, or a crash when trying to `.map()` over a raw string.

STRETCH GOALS

- Extract the load/save logic into a reusable `useLocalStorage` custom hook (a preview of Intermediate Chapter 4).
- Add a search box that filters the visible notes by title/content as you type.
- Store and display a "last edited" timestamp per note.
- Add a confirmation step before deleting a note, to avoid accidental data loss.
- Support basic Markdown-style formatting in the note body (e.g. rendering `**bold**` as bold text).

This project has no single solution file — the requirements above and your own judgment are the spec. Compare notes against the suggested component breakdown once you have something working, not before.

PROJECT 4 OF 7

Book Search with Routing and Debounced Search

PROJECT 4

Book Search with Routing and Debounced Search

A search results page and a separate detail page, navigated with React Router, with search-as-you-type that doesn't hammer the API

DIFFICULTY: ADVANCED

INTRODUCES: REACT ROUTER, DEBOUNCING

This project reaches slightly ahead of where the chapter-by-chapter courses are at — it needs two ideas that don't have their own Fundamentals chapter yet: **routing** (multiple "pages" in a single-page app) and **debouncing** (delaying a search request until typing actually pauses). Both get a working primer below; React Router gets its proper full chapter in Intermediate Chapter 5, and this project is a hands-on first taste of it ahead of that.

SKILLS THIS PROJECT EXERCISES

- React Router basics
- URL params (useParams)
- Debounced search
- useEffect cleanup
- Loading & error state
- List + detail view

REACT ROUTER — THE MINIMUM YOU NEED FOR THIS PROJECT

Install it with `npm install react-router-dom`. Four pieces cover this whole project: `<BrowserRouter>` wraps the app once, at the top; `<Routes>`/`<Route path="..." element={...}>` define which component renders for which URL; `<Link to="..."></Link>` navigates without a full page reload; and `useParams()` reads a dynamic piece of the URL (like a book's id) inside the component that route renders.

```
<Routes>
  <Route path="/" element={<SearchPage />} />
  <Route path="/book/:id" element={<BookDetailPage />} />
</Routes>
```

DEBOUNCING — THE MINIMUM YOU NEED FOR THIS PROJECT

Firing a fetch on every single keystroke wastes requests and can show results out of order as old, slow responses arrive after newer ones. Debouncing waits for a short pause in typing (e.g. 400ms with nothing new typed) before actually firing the request — using `setTimeout` inside a `useEffect`, with the cleanup function (from Fundamentals Chapter 8) cancelling the pending timeout if the user types again before it fires.

```
useEffect(() => {
  const timeoutId = setTimeout(() => {
    // the actual fetch goes here, using the current query value
  }, 400);

  return () => clearTimeout(timeoutId);
}, [query]);
```

Suggested Data Source

The [Open Library Search API] is free and needs no API key, similar to Open-Meteo in Project 2 — `https://openlibrary.org/search.json?q=<query>` returns a list of matching books, and `https://openlibrary.org/works/<id>.json` returns details for one specific book.

Requirements

- A search page at `/` with a text input — typing should debounce before firing a search request, not fire on every keystroke.
- Search results render as a list, each showing at least a title and author, and linking to a detail page for that specific book.
- A detail page at a URL like `/book/:id`, fetching and showing more information about that one specific book based on the id in the URL.
- The detail page includes a way to navigate back to the search page (a `<Link>`, not the browser's own back button only).
- Both pages handle loading and error states properly, reusing the status-based pattern from Project 2.
- Navigating between the two pages does not cause a full page reload — that's the whole point of using React Router instead of plain `<a>` tags.

Suggested Component Breakdown

```
App (BrowserRouter + Routes)
├── SearchPage              // route "/" — owns query state + debounced results
│   ├── SearchInput         // controlled text input
│   └── ResultsList         // maps results to ResultItem, each wrapped in a Link
│       └── ResultItem (×N)
└── BookDetailPage          // route "/book/:id" — reads id via useParams, fetches detail
```

A Reasonable Build Order

1. Set up `react-router-dom` with two placeholder page components and a working `<Link>` between them, before any fetching exists at all — confirm navigation itself works first.
2. Build the search input with a non-debounced fetch (firing on every keystroke) to confirm the API call and result rendering work correctly.
3. Convert the search effect to the debounced version, confirming with the browser's network tab that requests are no longer firing on every single keystroke.
4. Make each result item a ``<Link to={`/book/${id}`}>``, carrying the right id into the URL.
5. Build `BookDetailPage`, reading the id with `useParams()` and fetching that specific book's details in a `useEffect` keyed on the id.
6. Add loading/error handling to both pages last, once the core search → detail flow works end to end.

A STALE REQUEST CAN STILL "WIN" THE RACE

If the user types a new search before the previous request finishes, both requests are still in flight — and there's no strict guarantee the newer one resolves last. A query value captured in a ref, or an ignore-flag set in the effect's cleanup function, are both reasonable ways to discard a response that's no longer for the current search term. It's fine to skip this for a first pass and revisit it once the basic flow works.

STRETCH GOALS

- Extract the debounce logic into a reusable `useDebounce` custom hook (another preview of Intermediate Chapter 4).
- Add pagination or "load more" to the search results.

- Persist a small "recently viewed books" list to `localStorage`, shown on the search page.
- Show a loading skeleton instead of a plain "Loading..." message.
- Add a 404-style fallback route for any URL that doesn't match either page.

This project has no single solution file — the requirements above and your own judgment are the spec. Compare notes against the suggested component breakdown once you have something working, not before.

PROJECT 5 OF 7

Kanban Board with Drag-and-Drop

PROJECT 5

Kanban Board with Drag-and-Drop

Cards that move between columns by dragging, backed by a more complex piece of nested state

DIFFICULTY: ADVANCED

INTRODUCES: NATIVE HTML5 DRAG-AND-DROP

Every project so far has worked with a flat array — a list of todos, a list of notes. A Kanban board's state is one level more complex: an object where *each key holds its own array* (one per column), and the core interaction — dragging a card from one column to another — means immutably removing an item from one array and adding it to a different one, in the same update. This project also introduces the browser's native drag-and-drop events, which React doesn't wrap in anything special — they're used directly, the same way `onClick` always has been.

SKILLS THIS PROJECT EXERCISES

- Nested state shape
- Immutable array updates
- Native drag-and-drop events
- Conditional styling (drag-over)
- Lists + keys

THE NATIVE DRAG-AND-DROP EVENTS YOU'LL NEED

No library is required for the core requirement — five HTML attributes/events cover it: `draggable` on the card itself; `onDragStart` (fires when dragging begins — store which card and which column it came from); `onDragOver` on each column (must call `e.preventDefault()`, or the column will never accept a drop); and `onDrop` on each column (where the actual state update happens, moving the card into that column).

Requirements

- At least three columns (e.g. "To Do," "In Progress," "Done"), each showing its own list of cards.
- A way to add a new card with some text, landing in a specific column.
- Dragging a card from one column and dropping it on another moves it there — removed from the original column's list, added to the new one.
- A card can be deleted from whichever column it's currently in.

- The column currently being dragged over should look visually different (e.g. a highlighted border) while a card is hovering over it.
- Dropping a card back into its original column should not duplicate it or lose it.

Suggested Component Breakdown

```
App
└── KanbanBoard              // holds { todo: [...], inProgress: [...], done: [...] }
    ├── AddCardForm          // text input + column choice, adds to that column's array
    └── Column (×3)          // onDragOver / onDrop live here, plus drag-over highlight state
        └── Card (×N)        // draggable={true}, onDragStart, delete button
```

A reasonable shape for the state itself: `{ todo: [{ id, text }, ...], inProgress: [...], done: [...] }`. `KanbanBoard` owns this whole object — exactly the "lowest common ancestor" reasoning from Fundamentals Chapter 10, since moving a card between columns is fundamentally a change that affects two columns (siblings) at once, which only their shared parent can coordinate.

A Reasonable Build Order

1. Hardcode the columns object with a few sample cards in each, and get all three columns rendering their cards correctly with `.map()` before touching drag-and-drop at all.
2. Move the hardcoded object into `useState`, and build `AddCardForm` so new cards can be added to a chosen column's array.
3. Add `draggable` and `onDragStart` to `Card`, storing the card's id and its current column name (state, a ref, or `e.dataTransfer` all work) somewhere `onDrop` can read it from later.
4. Add `onDragOver` (with `e.preventDefault()`) and `onDrop` to `Column`. On drop, build a new columns object: filter the card out of its source column's array, and add it to the destination column's array.
5. Add the drag-over highlight — a small piece of state in each `Column` (or lifted up, tracking which column id is currently being dragged over) toggled by `onDragEnter` / `onDragLeave`.

6 Add the delete button per card last, once the drag-and-drop flow is solid.

TWO CLASSIC GOTCHAS

Forgetting `e.preventDefault()` inside `onDragOver` is the single most common reason a drop silently does nothing — the browser's default behavior for drag-over is to reject the drop entirely unless that default is explicitly cancelled. Separately, building the new columns object by mutating the existing arrays in place (e.g. `.push()` or `.splice()` directly on a state array) won't reliably trigger a re-render — always build new arrays (`.filter()`, spread) and a new outer object, the same immutability rule from Fundamentals Chapter 3, just one level deeper this time.

STRETCH GOALS

- Support reordering cards within the same column, not just moving between columns.
- Persist the whole board to `localStorage`, same pattern as Project 3's notes.
- Swap the native drag-and-drop API for a dedicated library (e.g. `@dnd-kit/core`), which adds proper keyboard/accessibility support that the native API lacks.
- Let columns themselves be added, renamed, or removed, rather than being fixed at three.
- Add a card-detail view (click to expand a card into a longer description, due date, etc.).

This project has no single solution file — the requirements above and your own judgment are the spec. Compare notes against the suggested component breakdown once you have something working, not before.

PROJECT 6 OF 7

Shopping Cart with Global State

PROJECT 6

Shopping Cart with Global State

A cart that needs to be readable and updatable from completely unrelated parts of the page — the real case for global state

DIFFICULTY: ADVANCED

INTRODUCES: THE CONTEXT API

Every previous project's state stayed within one small cluster of components, close enough together that lifting state up (Fundamentals Chapter 10) was a clean fit. A shopping cart is different: a product grid, a cart icon in the page header, and a full cart page all need to read or update the *same* cart — and they're nowhere near each other in the component tree. Lifting the cart all the way up to `App` and passing it down through every layer in between would mean prop drilling through components that have nothing to do with the cart at all. This is exactly the situation the **Context API** exists for, previewed here ahead of its full treatment in Intermediate Chapter 2.

SKILLS THIS PROJECT EXERCISES

- createContext / useContext
- Context Provider
- Derived values (totals, counts)
- Immutable array updates
- Avoiding prop drilling

THE CONTEXT API — THE MINIMUM YOU NEED FOR THIS PROJECT

`createContext()` creates a context object; wrapping part of the tree in `<CartContext.Provider value={...}>` makes that `value` readable by *any* component nested inside it, no matter how deep, via `useContext(CartContext)` — with no props passed through the components in between at all.

```
const CartContext = createContext(null);

function CartProvider({ children }) {
  const [items, setItems] = useState([]);
  // addToCart, removeFromCart, etc. defined here

  return (
    <CartContext.Provider value={{ items, addToCart, removeFromCart }}>
      {children}
    </CartContext.Provider>
  );
}

// anywhere deep inside <CartProvider>, with zero props passed down to reach it:
const { items, addToCart } = useContext(CartContext);
```

Requirements

- A product listing (hardcoded data is fine) with name, price, and an "Add to Cart" button on each.
- A persistent cart indicator (e.g. in a header) showing the total number of items, visible regardless of which "page" is currently shown.
- Adding the same product a second time increases its quantity in the cart, rather than creating a duplicate entry.
- A cart view listing every item with its quantity, a per-item subtotal, and a running total for the whole cart.
- A way to remove an item from the cart entirely, and a way to change an item's quantity directly.
- Every piece of this (product grid, header badge, cart view) reads from the same single source of cart state — no two components should ever show conflicting cart data.

Suggested Component Breakdown

```
App
└── CartProvider              // owns cart state + addToCart/removeFromCart/updateQuantity
    ├── Header
    │   └── CartBadge          // useContext — total item count
    ├── ProductList
    │   └── ProductCard (×N)   // useContext — just needs addToCart
```

```
    └── CartPage
        └── CartItem (×N)          // useContext — quantity, remove, subtotal
```

Notice that `Header` and `ProductList` are siblings, several layers removed from `CartPage` — exactly the layout where plain prop-passing would mean threading the cart through components that don't use it themselves. `CartProvider` wraps all three, and every consumer below it reaches the cart directly via `useContext`.

A Reasonable Build Order

1. Build the static product grid and a non-functional cart page shell first, with no Context at all yet.
2. Create `CartContext` and `CartProvider`, holding just the `items` array in `useState`, and wrap the whole app in it.
3. Write `addToCart`: check whether the product is already in the array; if so, increase that entry's quantity, otherwise add a new entry with quantity 1 — both cases building a new array, never mutating the existing one.
4. Wire `ProductCard`'s button to call `addToCart` via `useContext`, and confirm the cart's contents are correct by logging them, before building any UI for the cart itself.
5. Build `CartBadge` next — it only needs a derived total (sum of every item's quantity), read from the same context.
6. Build out the full `CartPage` / `CartItem` views last, adding remove and quantity-update actions to the context's value alongside `addToCart`.

THE DUPLICATE-ENTRY TRAP

The most common bug in a cart like this is `addToCart` always appending a new entry, so adding the same product three times produces three separate rows instead of one row with quantity 3. Always check first — find an existing entry with a matching product id, and only push a brand-new entry if none was found.

STRETCH GOALS

- Persist the cart to `localStorage`, same pattern as Project 3, so it survives a refresh.

- Replace the hand-rolled Context with a small state-management library — Zustand or Redux Toolkit, both covered properly in Advanced Chapter 1 — and compare how much boilerplate each removes versus plain Context.
- Add quantity +/- stepper buttons instead of a raw number input.
- Add a simple coupon-code field that applies a percentage discount to the total.
- Add a checkout confirmation step that clears the cart afterward.

This project has no single solution file — the requirements above and your own judgment are the spec. Compare notes against the suggested component breakdown once you have something working, not before.

PROJECT 7 OF 7

Dashboard Capstone — React Query, Next.js, Charts

PROJECT 7 · CAPSTONE

Dashboard Capstone — React Query, Next.js, Charts

Pulling together routing, server data, and visualization into one real-feeling application

DIFFICULTY: CAPSTONE

INTRODUCES: REACT QUERY, NEXT.JS, RECHARTS

This is the final project in the series — the first one to combine a server-data library, a different framework (Next.js instead of plain Vite), and a charting library, all in one build. Three brand-new tools are introduced below, each with its own primer; none of them have a full course chapter yet, since React Query and Next.js are both covered properly in Advanced Chapters 2 and 7.

SKILLS THIS PROJECT EXERCISES

- React Query (useQuery)
- Next.js file-based routing
- Dynamic routes
- Charting (Recharts)
- Loading/error states, simplified

NEXT.JS — THE MINIMUM YOU NEED FOR THIS PROJECT

Scaffold with `npx create-next-app@latest`. The headline difference from Vite: pages are defined by the **file system** rather than by code you write yourself — a file at `app/page.js` becomes the `/` route, and a folder named `app/coin/[id]/page.js` becomes a dynamic route matching `/coin/anything`, with `anything` readable inside that page via Next's routing hooks. Everything learned about components, props, and state still applies unchanged — Next.js only changes how a component becomes a "page."

REACT QUERY — THE MINIMUM YOU NEED FOR THIS PROJECT

Install with `npm install @tanstack/react-query`, and wrap the app once in a `<QueryClientProvider>`. From there, `useQuery` replaces the entire "`useState` + `useEffect` + manual loading/error flags" pattern from earlier projects in one hook call:

```
const { data, isLoading, isError } = useQuery({
  queryKey: ["coins"],
  queryFn: () => fetch("https://api.coingecko.com/api/v3/coins/markets?vs_currency=usd").then((r)
});
```

> React Query handles the loading/error state, caches the result under `queryKey`, and can automatically refetch it later — all without a single `useEffect` written by hand.

Suggested Data Source

The [CoinGecko API] is free and needs no API key for its public market-data endpoints — a good fit for a dashboard with multiple live-updating widgets and a price history chart per item.

Requirements

- An overview page listing several items (e.g. top cryptocurrencies) with at least name, current price, and 24h change, fetched with `useQuery`.
- Clicking an item navigates (via Next.js routing) to a detail page for that specific item, at a dynamic URL.
- The detail page fetches and renders a price-history chart using Recharts (a line chart is enough).
- Loading and error states on both pages are handled entirely through React Query's `isLoading` / `isError` flags — no manually-managed loading state alongside it.
- The overview page's data refreshes automatically after some interval (React Query's `refetchInterval` option) without a manual page reload.
- The layout reads reasonably on both a desktop-width and a narrow/mobile-width viewport.

Suggested Component Breakdown

```
app/layout.js                   // wraps everything in QueryClientProvider
app/page.js                     // route "/" — DashboardPage
└── DashboardPage
    └── CoinList                // useQuery — list of coins
        └── CoinCard (×N)       // Link to /coin/[id]
app/coin/[id]/page.js           // route "/coin/:id" — CoinDetailPage
└── CoinDetailPage
    └── PriceChart              // useQuery — price history; Recharts LineChart
```

A Reasonable Build Order

1. Scaffold the Next.js app and confirm the default starter page runs, before changing anything.
2. Install React Query, set up `QueryClientProvider` in the root layout, and get one `useQuery` call successfully logging fetched data to the console.
3. Build `CoinList` / `CoinCard`, rendering real data from that query, each card linking to its own dynamic detail route.
4. Build the dynamic route page, reading the id from the URL and firing a second `useQuery` for that specific item's history data.
5. Install Recharts and get a `<LineChart>` rendering the fetched price-history data on the detail page.
6. Add `refetchInterval` to the overview query last, and confirm (via the network tab) that it's actually refetching on its own over time.

DON'T RE-INTRODUCE MANUAL LOADING STATE ON TOP OF REACT QUERY

A common mistake when first adopting React Query is keeping an old habit alive — adding a separate `useState` for loading/error on top of the `isLoading` / `isError` React Query already provides. That duplicates logic React Query is specifically meant to remove; lean entirely on the flags `useQuery` returns instead.

STRETCH GOALS

- Add a Next.js API route as a thin proxy to the external API, avoiding any rate-limit/CORS concerns on the client.
- Add a search/filter box on the overview page.
- Add a second chart type (e.g. a bar chart comparing 24h change across items).
- Add a dark/light theme toggle, persisted across visits.
- Deploy the finished app to Vercel — Next.js's own hosting platform, built for exactly this kind of project.

This project has no single solution file — the requirements above and your own judgment are the spec. This also wraps up the React Projects course: seven briefs, beginner through capstone, covering everything from a first todo list to a real multi-tool dashboard.