LECTURE 29

Routing and API Integration

Giving a single-page app multiple "pages" — and connecting it to the real REST API you built in Express.

CSC336 Web Technologies

OVERVIEW

In This Lecture

1. Client-side routing with React Router: routes, Link, and route parameters
2. Nested routes, shared layouts, a not-found route, programmatic navigation, and protected routes
3. Fetching data with fetch or axios inside useEffect, and building a custom data hook
4. Handling loading and error states while consuming your REST API

React Router: Different URLs, No Reload

1. An SPA still needs multiple "pages" -- a home page, product details, a login page -- and real server data
2. Client-side routing makes URLs like /, /products, /products/12 show different content WITHOUT a new server request each time
3. Install with: npm install react-router-dom

Setting Up Routes

```
import { BrowserRouter, Routes, Route }
  from "react-router-dom";
createRoot(root).render(
  <BrowserRouter>
    <Routes>
      <Route path="/" element={<App />}>
        <Route index element={<Home />} />
        <Route path="products"
          element={<Products />} />
        <Route path="products/:id"
          element={<ProductDetails />} />
        <Route path="*" element={<NotFound />} />
      </Route>
    </Routes>
  </BrowserRouter>
);
```

Each <Route> maps a URL path to the component (element) that renders there -- React Router reads the current URL and renders only the match.

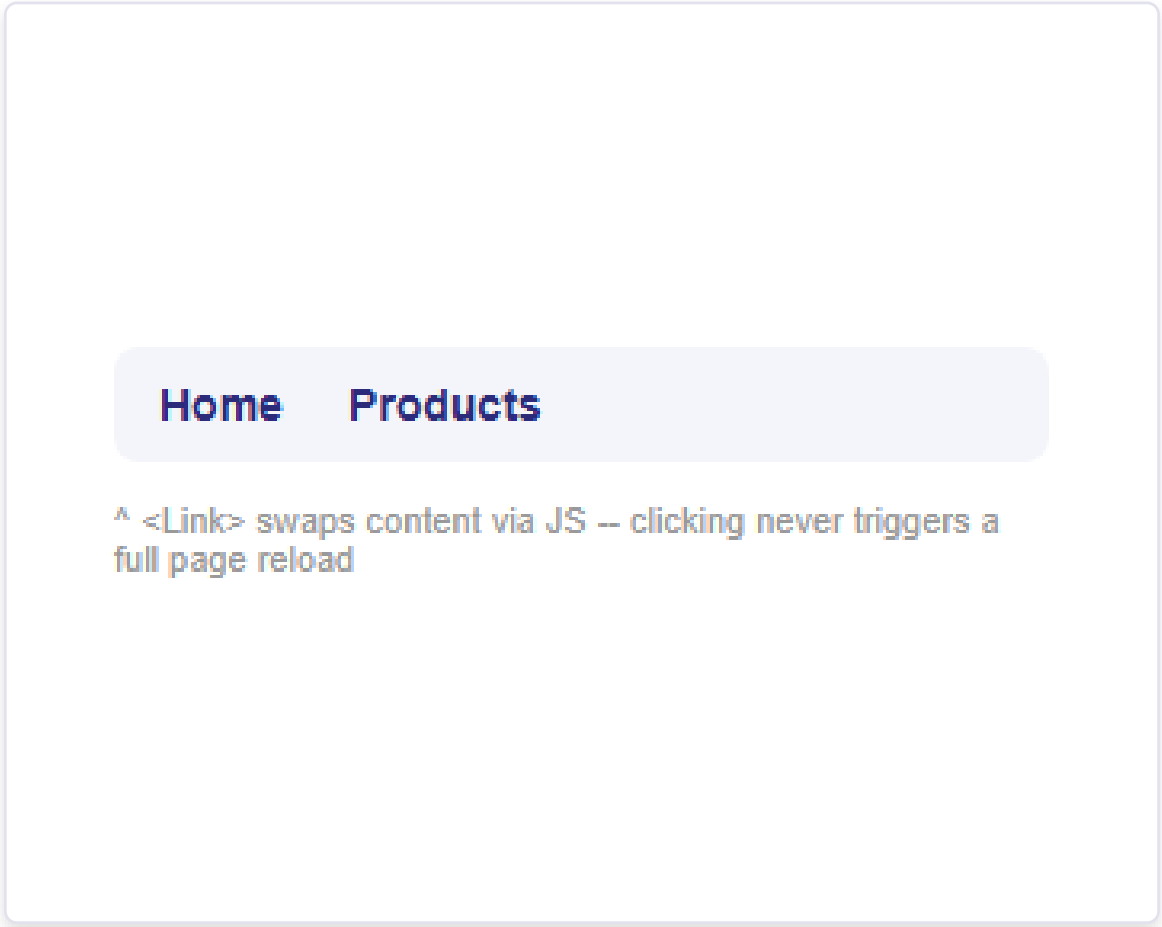
Never Use a Plain <a> Inside the App

1. A plain <a href="/products"> triggers a full browser page reload, defeating the purpose of an SPA

2. React Router's Link updates the URL and swaps content using JavaScript instead

A Navbar with Link

```
import { Link } from "react-router-dom";
function Navbar() {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/products">
        Products
      </Link>
    </nav>
  );
}
```



NavLink for Active-Link Styling

TIP

NavLink behaves like Link but automatically adds an active CSS class (or lets you compute a class/style) when its to path matches the current URL -- handy for highlighting the current page in a navbar.

Dynamic URL Segments

1. A route parameter is a dynamic segment written with a leading colon, like :id in products/:id
2. Read it inside the matching component with the useParams hook
3. Route params are always strings -- convert with Number(id) if you need a number

Reading :id with useParams

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

function ProductDetails() {
  const { id } = useParams();
  return <p>Showing details for #{id}</p>;
}
```

Visiting /products/12 renders ProductDetails with id equal to the string "12".

REFERENCE

React Router API at a Glance

API	Purpose
<Link to="...">	Client-side navigation, no page reload
<NavLink to="...">	Like Link, plus an automatic "active" class
useParams()	Reads dynamic route segments like :id
useNavigate()	Triggers navigation from inside your own code
<Navigate to="..." />	Redirects declaratively (e.g. from a protected route)
<Outlet />	Renders the matching nested route inside a layout

A Shared Layout via <Outlet />

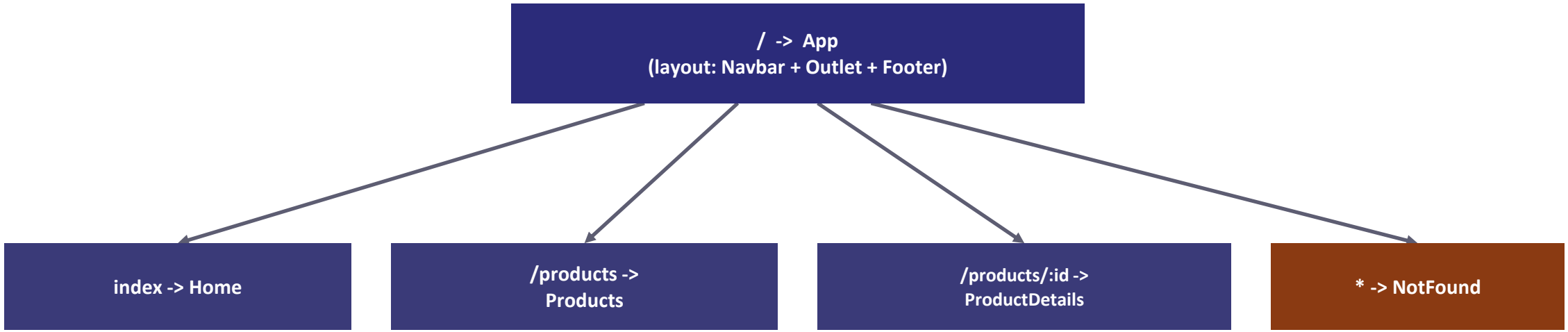
1. Home, Products, and ProductDetails are declared INSIDE the / route for App -- a nested route
2. App acts as a shared layout component (Navbar + Footer), rendering its nested route's content wherever it places an <Outlet />
3. This avoids repeating shared UI in every single page component

App as a Layout Component

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

function App() {
  return (
    <div>
      <Navbar />
      <main>
        <Outlet />
      </main>
      <Footer />
    </div>
  );
}
```

How One URL Tree Maps to Components



Navbar and Footer render on every page; <Outlet /> is replaced with Home, Products, or ProductDetails depending on the current URL.

A Catch-All 404 Route

```
function NotFound() {
  return <h1>404 -- Page not found</h1>;
}

// Placed LAST in the route list:
<Route path="*" element={<NotFound />} />
```

The special path "*" matches any URL that didn't match an earlier route -- placing it last ensures it only catches what's left.

Navigating From Code, Not a Click

1. Sometimes you need to navigate in response to code -- e.g. redirecting after a successful form submission

2. The useNavigate hook returns a function you call for this

Redirecting After Login

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

function LoginForm() {
  const navigate = useNavigate();

  async function handleSubmit(e) {
    e.preventDefault();
    // ...perform login...
    navigate("/dashboard");
  }

  return <form onSubmit={handleSubmit}>...</form>;
}
```

Gating Content Behind Login

1. A protected route only renders its content for authenticated users, redirecting anyone else

2. A common pattern: a small wrapper component checking an auth hook/context

A ProtectedRoute Wrapper

```
function ProtectedRoute({ children }) {
  const { user } = useAuth();
  if (!user) {
    return <Navigate to="/login" replace />;
  }
  return children;
}
```

Wrap any route needing login: element={<ProtectedRoute><Dashboard /></ProtectedRoute>}. replace keeps the blocked page out of browser history.

Data Fetching Is a Side Effect

1. In Lecture 25 you built a REST API with Express -- e.g. GET /api/products and GET /api/products/:id

2. Calling that API reaches OUTSIDE the component, so it belongs inside useEffect (Lecture 28)

Using fetch

```
function Products() {
  const [products, setProducts] =
    useState([]);
  useEffect(() => {
    fetch("/api/products")
      .then((res) => res.json())
      .then((data) => setProducts(data));
  }, []);
  return (
    <ul>
      {products.map((p) => (
        <li key={p._id}>{p.name}</li>
      ))}
    </ul>
  );
}
```

Using axios

```
import axios from "axios";
function Products() {
  const [products, setProducts] =
    useState([]);
  useEffect(() => {
    axios.get("/api/products")
      .then((res) => setProducts(res.data));
  }, []);
  return (
    <ul>
      {products.map((p) => (
        <li key={p._id}>{p.name}</li>
      ))}
    </ul>
  );
}
```

Why axios Is Often Preferred

1. Automatically parses JSON responses -- no manual res.json() step
2. A slightly simpler API for the common GET/POST/etc. calls
3. More consistent error handling: network AND non-2xx errors both land in .catch, unlike fetch (where a 404/500 still resolves and must be checked manually)

Track Three Pieces of State, Not One

1. A real request takes time and can fail -- from a network problem or a server error status

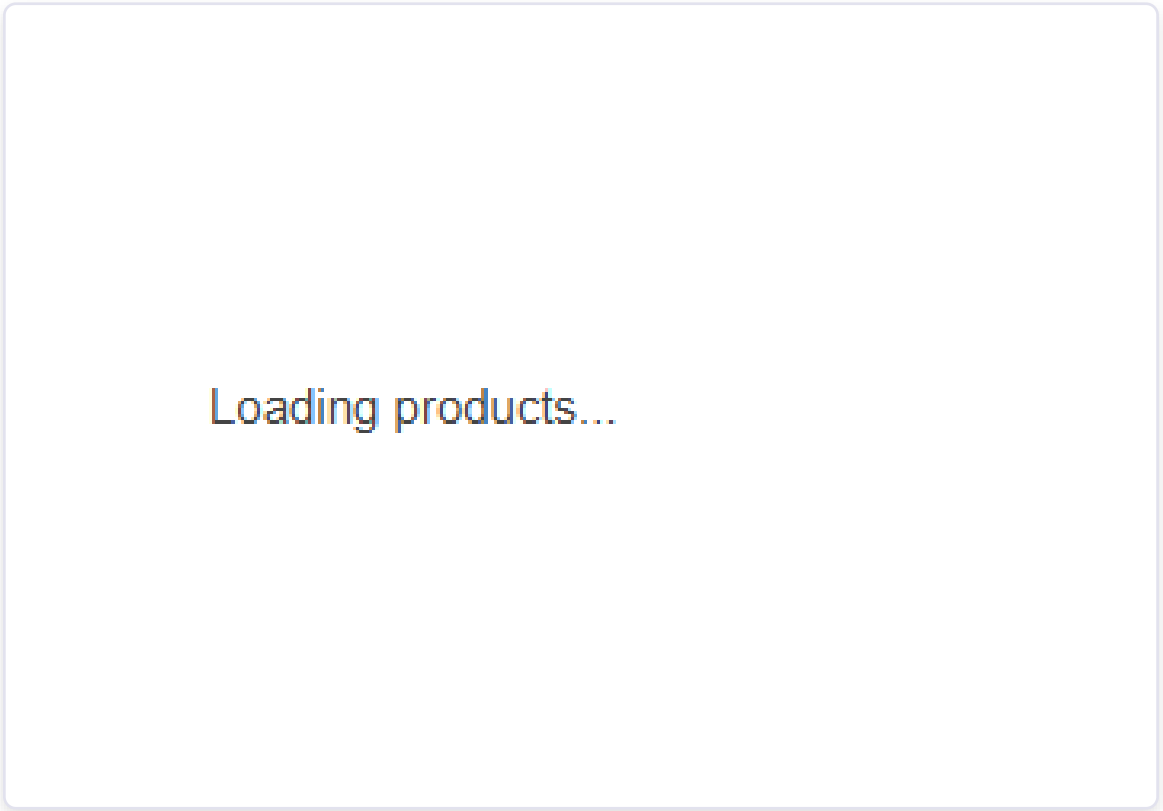
2. Track the data itself, a loading flag, and an error, and reflect all three in the UI

Products, Fully Handled

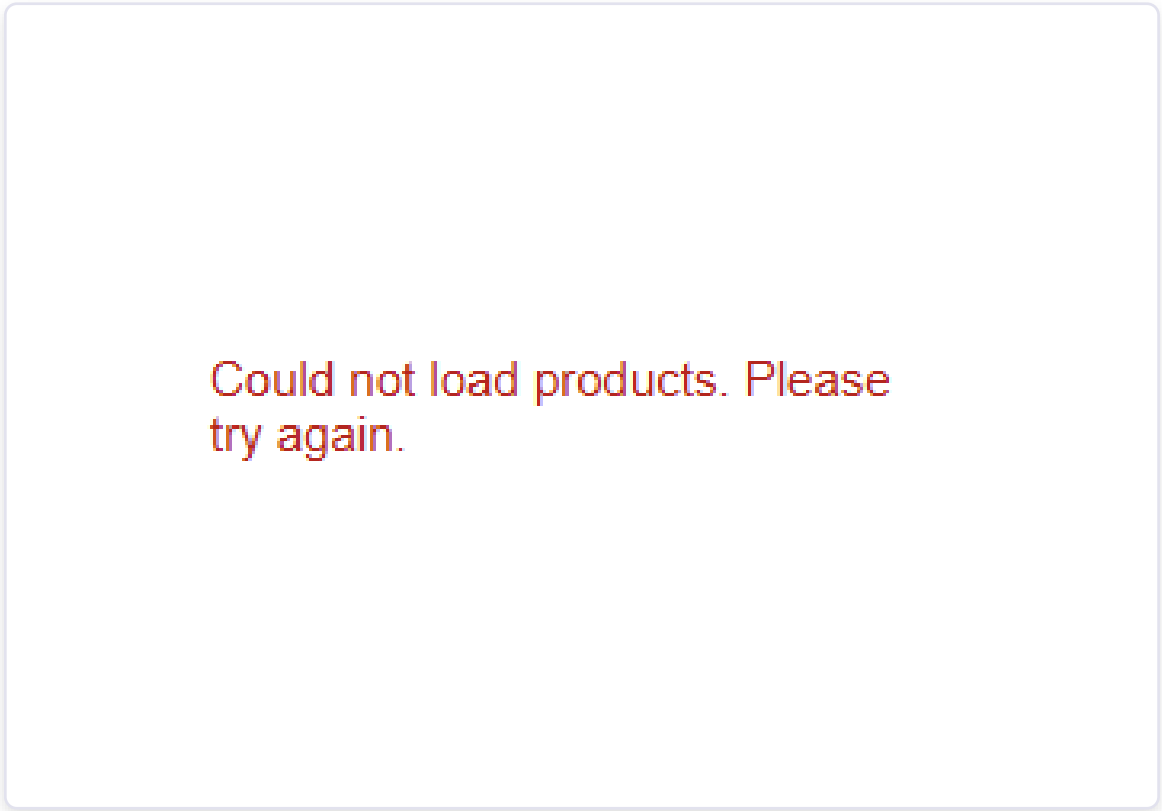
```
function Products() {
  const [products, setProducts] = useState([]);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState(null);
  useEffect(() => {
    axios.get("/api/products")
      .then((res) => setProducts(res.data))
      .catch(() => setError("Could not load."))
      .finally(() => setIsLoading(false));
  }, []);
  if (isLoading) return <p>Loading...</p>;
  if (error) return <p role="alert">{error}</p>;
  return <ul>{/* render products */}</ul>;
}
```

Two States, Rendered

ISLOADING: TRUE



ERROR: SET

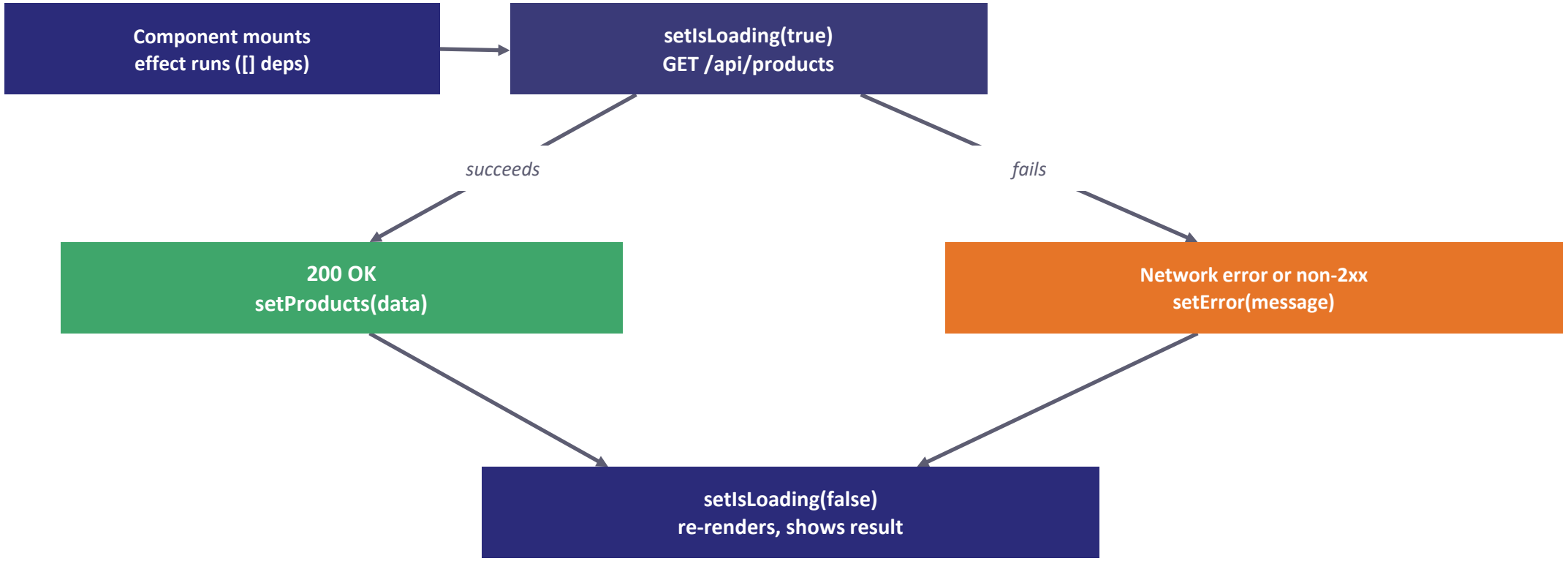


The Success State, Rendered

Once the request resolves and isLoading is false with no error, the real list renders.

- Keyboard — $45
- Mouse — $20
- Monitor — $199

The Full Request Lifecycle



Both branches funnel through the same finally: setIsLoading(false) always runs, so the UI never gets stuck showing "Loading...".

"Loading, Error, Data" Shows Up Everywhere

1. This exact pattern repeats in almost every component that talks to an API

2. A great candidate for a custom hook (Lecture 28), so the logic isn't rewritten each time

useFetch.js

```
function useFetch(url) {
  const [data, setData] = useState(null);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState(null);
  useEffect(() => {
    let ignore = false;
    axios.get(url)
      .then((res) => { if (!ignore) setData(res.data); })
      .catch(() => { if (!ignore) setError("Failed."); })
      .finally(() => { if (!ignore) setIsLoading(false); });
    return () => { ignore = true; };
  }, [url]);
  return { data, isLoading, error };
}
```

The ignore flag inside cleanup avoids updating state from a late response after the component has already unmounted.

CUSTOM DATA HOOK

Any Endpoint, One Line

```
function Products() {
  const { data, isLoading, error } =
    useFetch("/api/products");
  if (isLoading) return <p>Loading...</p>;
  if (error) return <p role="alert">{error}</p>;
  return <ul>{/* data.map(...) */}</ul>;
}
function ProductDetails() {
  const { id } = useParams();
  const { data, isLoading, error } =
    useFetch(`/api/products/${id}`);
  if (isLoading) return <p>Loading...</p>;
  if (error) return <p role="alert">{error}</p>;
  return <h1>{data.name}</h1>;
}
```

Combining route parameters (useParams) with useFetch is exactly how a real product-details page, backed by GET /api/products/:id, gets built.

CORS During Local Development

WARNING

If your Express API (e.g. localhost:5000) and your Vite dev server (e.g. localhost:5173) run on different ports, the browser treats them as different origins. Make sure Express uses the cors middleware (Lecture 25) so the browser allows these requests during development.

KEY TAKEAWAYS

Lecture 29 in Seven Points

1. React Router enables client-side routing: URLs map to components without a full page reload; use Link, never <a>, for in-app navigation.
2. Route parameters (:id) capture dynamic URL segments, read with useParams.
3. Nested routes with <Outlet /> share a layout (navbar, footer) across pages; a path="*" route provides a not-found page.
4. useNavigate triggers navigation from code; protected routes redirect unauthenticated users away with <Navigate replace />.
5. Data fetching (fetch or axios) belongs inside useEffect, since it is a side effect; axios adds automatic JSON parsing and simpler error handling.
6. Always track loading and error state alongside your data, and show the user feedback for every case, not just success.
7. A custom useFetch hook removes repetition when many components need to load data from your REST API.