Project Starter
Frontend Source Code Explanation

App.js: The top-level component (the “composition root”)

What App.js is responsible for

App.js is the main component that:

1. **Wraps the whole UI with global providers** (here: the shopping cart context)
2. **Shows components that should appear on every page** (here: the Header)
3. **Defines routing** (which page component is shown for each URL)

The Code

```
import React from "react";
import Header from "./components/Header";
import Home from "./pages/Home";
import Checkout from "./pages/Checkout";
import ProductDetail from "./pages/ProductDetail";
import { CartProvider } from "./context/CartContext";
import { Routes, Route } from "react-router-dom";

import "./App.css";

function App() {
  return (
    <CartProvider>
      <Header />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/checkout" element={<Checkout />} />
        <Route path="/products/:id" element={<ProductDetail />} />
      </Routes>
    </CartProvider>
  );
}

export default App;
```

Line-by-line explanation

Imports

```
import React from "react";
```

- Imports React so we can write JSX like `<Header />`.

```
import Header from "./components/Header";
```

- `Header` is a reusable component (navigation bar, cart icon, etc.)
- It appears on every page.

```
import Home from "./pages/Home";
import Checkout from "./pages/Checkout";
import ProductDetail from "./pages/ProductDetail";
```

- These are **page components**.
- A common convention:
    - `components/` → smaller reusable UI pieces
    - `pages/` → full “screens” that match routes

```
import { CartProvider } from "./context/CartContext";
```

- This imports the **Context Provider** for the shopping cart.
- Anything inside `<CartProvider>` can access cart state with `useCart()` or `useContext(CartContext)`.

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

- These come from **React Router v6**.
- They let us map URLs (paths) to page components.

```
import "./App.css";
```

- Loads app-level CSS (global layout tweaks, background, etc.)

The App component

```
function App() {
  return (
    <CartProvider>
      <Header />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/checkout" element={<Checkout />} />
        <Route path="/products/:id" element={<ProductDetail />} />
      </Routes>
    </CartProvider>
  );
}
```

1) `<CartProvider> ... </CartProvider>`

- This is a **global state wrapper**.
- It provides cart state and cart functions (add, remove, increment, etc.) to:
  - `Header` (to show cart badge count)
  - `Home` (add to cart buttons)
  - `Checkout` (show items and quantities)
  - `ProductDetail` (add to cart button)

Why it's here (top-level)?

- Because multiple pages and components need the cart.
- If we placed it only inside `Checkout`, `Home` and `Header` couldn't access the cart.

---

2) `<Header />`

- Rendered **outside** `<Routes>`, meaning:
  - Header is visible on every page.
- This is typical for layouts: header/navbar/footer remain constant while main content changes.

---

3) `<Routes>` and `<Route>`

This defines the navigation rules of the app.

Route 1: Home page

```
<Route path="/" element={<Home />} />
```

- When URL is `http://localhost:3000/`
- React Router shows the `<Home />` page.

Route 2: Checkout page

```
<Route path="/checkout" element={<Checkout />} />
```

- When URL is `http://localhost:3000/checkout`
- It shows the `<Checkout />` page.

Route 3: Product detail page (dynamic route)

```
<Route path="/products/:id" element={<ProductDetail />} />
```

- `:id` is a **route parameter** (dynamic value).
- Examples:
    - /products/1
    - /products/5

Inside `ProductDetail`, we read that `id` using:

```
const { id } = useParams();
```

Then we can call the backend:

- GET /api/products/{id}

Key points:

1. **App.js is the app's "assembly file"**
    - It connects providers + shared layout + routing.
2. **Providers go high in the tree**
    - If many pages need a state (cart), wrap it at the top.
3. **Header outside Routes = shown on every page**
    - Great for consistent navigation.
4. **React Router v6 uses `element={<Component />}`**
    - (Different from older v5 syntax)
5. **`:id` means dynamic routing**
    - Enables detail pages like `/products/5`.

src/index.js (React App Entry Point)

What is index.js?

index.js is the **starting point** of the React frontend. It is the file that:

1. Finds the HTML element where React will "mount"
2. Creates the React root (React 18 style)
3. Renders the entire app (<App />)
4. Wraps the app with global providers (Theme, Router, etc.)

In simple terms:

index.js is where React is attached to the web page.

The code (for reference)

```
import React from "react";
import { createRoot } from "react-dom/client";
import App from "./App";
import { ThemeProvider } from "@mui/material/styles";
import CssBaseline from "@mui/material/CssBaseline";
import theme from "./theme";
import { BrowserRouter } from "react-router-dom";

const container = document.getElementById("root");
const root = createRoot(container);

root.render(
  <React.StrictMode>
    <ThemeProvider theme={theme}>
      <CssBaseline />
      <BrowserRouter>
        <App />
      </BrowserRouter>
    </ThemeProvider>
  </React.StrictMode>
);
```

Step-by-step explanation

1) Imports

React

import React from "react";

- Needed to write JSX and use React features.

React 18 root API

import { createRoot } from "react-dom/client";

- This is the React 18 way of rendering an app.
- Older projects used ReactDOM.render(...) (React 17 and earlier).

The App component

import App from "./App";

- <App /> is the main component of your UI.
- It contains the header and the route definitions.

Material UI theme support

import { ThemeProvider } from "@mui/material/styles";

import CssBaseline from "@mui/material/CssBaseline";

import theme from "./theme";

- **ThemeProvider**: makes a theme available to all Material UI components.
- **theme**: your custom theme configuration (colors, typography, spacing, etc.)
- **CssBaseline**: a Material UI component that applies a consistent "base CSS reset"
    - Removes browser inconsistencies
    - Gives clean default styling

React Router support

import { BrowserRouter } from "react-router-dom";

- Wraps the app so that routing works.
- Enables:
    - <Routes>, <Route>
    - <Link to="/...">
    - useParams(), useNavigate()

Without <BrowserRouter>, you get errors like:

"You should not use <Routes> outside a <Router>"

---

2) Find the HTML mount point

```
const container = document.getElementById("root");
```

In public/index.html, you have something like:

```
<div id="root"></div>
```

React needs a real DOM element to “attach” the whole application.

So:

- container is the <div id="root">...</div>
- Everything React renders will appear inside that element.

---

3) Create a React root (React 18)

```
const root = createRoot(container);
```

This creates a React rendering root.
React then controls everything inside that container.

Key point:

- createRoot is the modern rendering entry point.
- It enables newer React features and optimizations.

---

4) Render the application (with global wrappers)

```
root.render(

  <React.StrictMode>

    <ThemeProvider theme={theme}>

      <CssBaseline />

      <BrowserRouter>

        <App />

      </BrowserRouter>
    </ThemeProvider>
  </React.StrictMode>
);
```

This is the most important part. It says:

"Render App, but wrap it with StrictMode, ThemeProvider, CssBaseline, and BrowserRouter."

A) <React.StrictMode>

- A development-only wrapper that helps find bugs.
- It can cause some lifecycle methods to run twice in development (not production).
- It encourages best practices and warns about unsafe patterns.

B) <ThemeProvider theme={theme}>

- Provides your Material UI theme globally.
- Every MUI component (Button, AppBar, Typography, etc.) can use the theme.

Typical benefits:

- Consistent colors and typography
- One place to change design style

C) <CssBaseline />

- Applies global baseline CSS styles (like a reset + good defaults).
- This prevents different browsers from rendering components with inconsistent default margins/fonts.

D) <BrowserRouter>

- Enables client-side routing (single page application behavior).
- Without reloading the entire page, React changes content based on URL.

Example:

- / shows Home
- /products/5 shows ProductDetail
- /checkout shows Checkout

Key points

1. **index.js mounts React into the HTML page**
    - document.getElementById("root")
2. **React 18 uses createRoot**
    - not ReactDOM.render
3. **Global wrappers go here**
    - Theme provider
    - Router provider
    - Other providers (auth, Redux, Cart, etc.)
4. **BrowserRouter is required for routing**
    - Enables navigation without full page refresh
5. **Material UI needs ThemeProvider and CssBaseline**
    - For consistent design and styling defaults

src/theme.js (Material UI Theme)

What is theme.js?

theme.js defines the **global design system** for your React application when using **Material UI (MUI)**.

Instead of manually setting colors and styles on every component, you define a theme once, and MUI components automatically follow it.

In your project, ThemeProvider in index.js applies this theme to the whole app.

The code (reference)

```
import { createTheme } from '@mui/material/styles';

const theme = createTheme({
  palette: {
    primary: { main: '#202529ff' },
    secondary: { main: '#f50057' }
  },
  components: {
    MuiAppBar: {
      defaultProps: {
        elevation: 1
      }
    }
  }
});

export default theme;
```

Step-by-step explanation

1) Import createTheme

import { createTheme } from '@mui/material/styles';

- createTheme() is a Material UI function that builds a **theme object**.
- That theme object will be passed into <ThemeProvider theme={theme}> so MUI can style all components consistently.

2) Create the theme object

```
const theme = createTheme({
  ...
});
```

This object contains **configuration sections**, such as:

- `palette` → colors
- `typography` → fonts & sizes (not used here, but common)
- `components` → per-component default settings and style overrides

3) palette: your main colors

```
palette: {
  primary: { main: '#202529ff' },
  secondary: { main: '#f50057' }
},
```

What is primary?

- The *primary color* is the default highlight color for the app.
- Used automatically by many components when you write:

```
<Button variant="contained" color="primary">Buy</Button>
```

or:

```
<AppBar color="primary">...</AppBar>
```

Here:

- `primary.main = '#202529ff'`
  That's a dark, near-black color (often used for modern headers/navbars).

What is secondary?

- A second accent color used for contrast or special UI elements like badges, highlights, etc.
- Example:

```
<Badge color="secondary" badgeContent={3}>...</Badge>
```

Here:

- `secondary.main = '#f50057'`
  A bright pink accent.

Why use palette instead of hardcoding colors?

- Consistency: same colors everywhere
- Maintainability: change one line → whole UI updates
- Cleaner code: components use `color="primary"` instead of manual CSS

---

4) components: global component configuration

```
components: {
  MuiAppBar: {
    defaultProps: {
      elevation: 1
    }
  }
}
```

This section lets you set:

- `defaultProps` → default values for props
- `styleOverrides` → override CSS styles globally
- `variants` → define new component variants

MuiAppBar.defaultProps.elevation = 1

- `AppBar` uses the concept of "elevation" (shadow depth).
- `elevation: 0` = no shadow
- `elevation: 1` = small shadow
- higher numbers = stronger shadow

With this theme config:

Every `<AppBar />` in the whole app will automatically have a small shadow (elevation 1), unless you override it locally.

This is great because you don't need to repeat:

```
<AppBar elevation={1} />
```

in every file.

---

5) Export the theme

```
export default theme;
```

This allows `index.js` to import and apply it:

```
<ThemeProvider theme={theme}>
  ...
</ThemeProvider>
```

Key points:

1. **The theme is the “design rules” of the app**
   - Colors, fonts, spacing, component defaults.
2. **`palette` defines global color identity**
   - `primary` and `secondary` are used automatically by MUI components.
3. **`components` section lets you control MUI globally**
   - Set default props for consistency.
   - Override styles once instead of repeating.
4. **Maintainability benefit**
   - Changing colors and common styles becomes easy and centralized.

src/config.js (Centralized Backend Configuration)

What is config.js?

config.js is a small but very important file that centralizes **all backend URLs** used by the frontend.

Main goals:

1. **Single source of truth** for the backend base URL
2. Avoid hard-coding http://localhost:8080 in many files
3. Make it easy to change environments (development vs production)
4. Provide helper utilities (like building image URLs)

The code (reference)

```
// src/config.js

// Backend base URL (only change here or via .env)
export const BACKEND_BASE_URL =
  process.env.REACT_APP_BACKEND_URL || 'http://localhost:8080';

// API base URL
export const API_BASE_URL = `${BACKEND_BASE_URL}/api`;

// Helper: build full image URL from DB value
export const resolveImageUrl = (pathOrUrl) => {
  if (!pathOrUrl) return '';

  // If it's already a full URL (http/https), just return it
  if (/^https?:\/\//i.test(pathOrUrl)) {
    return pathOrUrl;
  }

  // Ensure it starts with a single slash
  if (!pathOrUrl.startsWith('/')) {
    pathOrUrl = `/${pathOrUrl}`;
  }

  return `${BACKEND_BASE_URL}${pathOrUrl}`;
};
```

---

1) BACKEND_BASE_URL: the backend address (one place to change)

```
export const BACKEND_BASE_URL =
  process.env.REACT_APP_BACKEND_URL || 'http://localhost:8080';
```

What it does

This defines the **backend server URL**.

- First option: process.env.REACT_APP_BACKEND_URL
    - This reads from a .env file (or system environment variables).
- Fallback option: 'http://localhost:8080'
    - Used if the environment variable is not set.

Why is this useful?

Because in real projects, backend URL changes depending on where the app runs:

- Development:
    - http://localhost:8080
- Production:
    - https://api.myshop.com

With this file, you change it **in one place**.

Important note (React env variables)

In Create React App, environment variables must start with REACT_APP_.

Example frontend/.env:

```
REACT_APP_BACKEND_URL=http://localhost:8080
```

---

2) API_BASE_URL: base URL for REST endpoints

```
export const API_BASE_URL = `${BACKEND_BASE_URL}/api`;
```

What it does

Most backend endpoints are under /api, for example:

- GET /api/products
- GET /api/products/top
- GET /api/products/5

So instead of writing the full URL every time, we define the API base once.

Example:

- If BACKEND_BASE_URL = http://localhost:8080
- then API_BASE_URL = http://localhost:8080/api

This is typically used by Axios:

```
axios.create({ baseURL: API_BASE_URL })
```

---

3) resolveImageUrl: convert DB image paths into real URLs

```
export const resolveImageUrl = (pathOrUrl) => {
  ...
};
```

Why do we need this?

In the database, we store only a **path**, like:

/images/coffee.jpg

But the browser needs a **full URL**, like:

http://localhost:8080/images/coffee.jpg

So resolveImageUrl() converts what comes from DB into a usable URL.

Step-by-step inside resolveImageUrl

A) Handle missing values

```
if (!pathOrUrl) return '';
```

If the product has no image URL, we return an empty string to avoid errors.

B) If the value is already a full URL, return it

```
if (/^https?:\/\//i.test(pathOrUrl)) {
  return pathOrUrl;
}
```

This checks whether the string starts with:

- http:// or https://

If yes, we don’t modify it.

Why is this useful?
Because later you might use a CDN and store full URLs like:

https://cdn.myshop.com/images/coffee.jpg

This function supports both cases:

- DB stores a path (/images/...)
- DB stores a full URL (https://...)

C) Ensure the path starts with /

```
if (!pathOrUrl.startsWith('/')) {
  pathOrUrl = `/${pathOrUrl}`;
}
```

Sometimes someone might store:

- images/coffee.jpg (missing slash)

This fixes it automatically.

D) Combine backend URL + path

```
return `${BACKEND_BASE_URL}${pathOrUrl}`;
```

So:

- BACKEND_BASE_URL = http://localhost:8080
- pathOrUrl = /images/coffee.jpg
  → final result:
- http://localhost:8080/images/coffee.jpg

Key points:

1. **Centralize configuration**
   - Never hardcode backend URLs in many components.
2. **Support multiple environments**
   - .env for development, deployment config for production.

3. **Separate API URLs from static file URLs**
    - API: .../api/...
    - Images/static: .../images/...
4. **Write small helper functions**
    - resolveImageUrl() avoids repeated string logic everywhere.

src/api/axios.js (Axios Client Configuration)

What is this file for?

axios.js creates a **pre-configured Axios instance** that the whole frontend uses to call the backend API.

Instead of writing this in every component:

axios.get("http://localhost:8080/api/products")

we create one reusable client:

api.get("/products")

This makes the code cleaner and easier to maintain.

The code (reference)

```
// src/api/axios.js
import axios from 'axios';
import { API_BASE_URL } from '../config';

const api = axios.create({
  baseURL: API_BASE_URL,
  timeout: 5000,
});

export default api;
```

Line-by-line explanation

1) Import Axios

import axios from 'axios';

- Axios is an HTTP client library.
- It can send requests like GET, POST, PUT, DELETE.
- It automatically parses JSON responses into JavaScript objects (res.data).

---

2) Import the API base URL

import { API_BASE_URL } from '../config';

- API_BASE_URL is the *backend API prefix*, such as:
    - http://localhost:8080/api
- It is defined in config.js, so you change it in one place.

This avoids hardcoding URLs throughout the project.

---

3) Create a custom Axios instance

```
const api = axios.create({
  baseURL: API_BASE_URL,
  timeout: 5000,
});
```

This creates an Axios "client object" with default settings.

baseURL: API_BASE_URL

- Automatically prepends the base URL to all requests.

Example:

```
api.get("/products");
```

actually calls:

```
GET http://localhost:8080/api/products
```

So your code stays short and consistent.

timeout: 5000

- If the server does not respond within **5000 ms (5 seconds)**, Axios rejects the request with a timeout error.
- This prevents the UI from "hanging forever" if the backend is down or very slow.

In real apps you might adjust this (e.g., 10 seconds), but 5 seconds is a good teaching default.

---

4) Export the Axios instance

```
export default api;
```

Now any part of the frontend can do:

```
import api from "../api/axios";


const res = await api.get("/products");
```

How this should be used (best practice)

Instead of using api directly inside every component, you typically put API calls into **service modules**:

Example: src/services/productService.js

```
import api from "../api/axios";


export const getAllProducts = async () => {
  const res = await api.get("/products");
  return res.data;
};
```

Then components simply call service functions and focus on UI.

Important technique (for later): interceptors

A big reason we create a single Axios instance is that later you can add **interceptors**:

- Attach authentication tokens to every request
- Handle errors globally (401, 500, etc.)

Example idea (not required yet):

```
api.interceptors.request.use((config) => {
  const token = localStorage.getItem("token");
  if (token) config.headers.Authorization = `Bearer ${token}`;
  return config;
});
```

Key takeaway:

- axios.js provides a **single, shared HTTP client**
- It centralizes:
  - API base URL
  - request timeouts
  - (later) authentication and global error handling
- It keeps components clean and maintainable

---

productService.js (Service Layer for API Calls)

What is productService.js?

productService.js is part of the **service layer** in the frontend.

Its job is to:

- Collect all product-related API requests in **one place**
- Keep React components (UI pages) clean and focused on rendering
- Hide HTTP details (URLs, Axios calls, etc.) from the rest of the app

So instead of writing Axios calls inside Home.js, Checkout.js, ProductDetail.js, we write them once here and reuse them.

The code (reference)

```
import api from "../api/axios";

export const getTopProducts = async () => {
  const res = await api.get("/products/top");
  return res.data;
};

export const getAllProducts = async () => {
  const res = await api.get("/products");
  return res.data;
};

export const getProductById = async (id) => {
  const res = await api.get(`/products/${id}`);
  return res.data;
};
```

Line-by-line explanation

1) Import the configured Axios client

import api from "../api/axios";

- api is the Axios instance you created in axios.js.
- It already knows the base URL:
  - baseURL = API_BASE_URL
  - e.g. http://localhost:8080/api

- That means every request here can use short paths like "/products".

Function 1: getTopProducts()

```
export const getTopProducts = async () => {
  const res = await api.get("/products/top");
  return res.data;
};
```

What it does

- Calls backend endpoint:
  - GET /api/products/top
- The backend returns a JSON array of featured products (in your project).

Why return res.data?

Axios response looks like:

```
{
  data: ...,      // JSON payload you care about
  status: 200,
  headers: ...,
  config: ...,
}
```

Most of the time, the UI only needs the JSON payload (data), so we return that directly.

Benefit: Components can do:

```
const products = await getTopProducts();
```

without worrying about Axios response structure.

Function 2: getAllProducts()

```
export const getAllProducts = async () => {
  const res = await api.get("/products");
  return res.data;
};
```

What it does

- Calls:
    - GET /api/products
- Returns all products in the database.

Used for:

- Full product list pages
- Admin listing (later)
- Search/filter pages (later)

Function 3: getProductById(id)

```
export const getProductById = async (id) => {
  const res = await api.get(`/products/${id}`);
  return res.data;
};
```

What it does

- Calls:
    - GET /api/products/{id}
- Example:
    - getProductById(5) → GET /api/products/5

This is used by ProductDetail.js, where the id comes from the route:

```
const { id } = useParams();
```

Why backticks (template string)?

Because we need to insert a variable into the URL path:

- `/products/${id}` creates a string like "/products/5"

Why a "service layer" is good architecture (frontend version)

Benefits

1. **Single source of truth for API endpoints**
    - If backend URL changes (/products/top becomes /products/featured) you update it once here, not in many components.
2. **Cleaner components**
    - UI files focus on layout, state, and rendering—not networking code.
3. **Better testing and reuse**
    - You can mock service functions easily in tests.
    - Multiple pages can reuse the same functions.
4. **Easy to extend**
    - Add more functions like:
        - createProduct(product)
        - updateProduct(id, product)
        - deleteProduct(id)
        - searchProducts(query)

Typical usage in a React page (example)

In Home.js:

```
useEffect(() => {
  getTopProducts()
    .then(setProducts)
    .catch(console.error);
}, []);
```

Notice how Home.js doesn't know about axios, baseURL, or HTTP details.
It just calls a function that returns product data.

Key points:

- **Service layer = "API functions"**
- It separates **data fetching logic** from **UI rendering**
- Makes the project easier to maintain as it grows

components/Header.js (Navigation + Cart Badge)

What is Header.js responsible for?

Header.js renders the top navigation bar of the app. It:

1. Shows the **site title** (“Ecomm Demo”) and links it to the Home page
2. Shows a **shopping cart icon**
3. Displays the **total number of items in the cart** using a badge
4. Links the cart icon to the **Checkout page**

This header appears on every page because it’s rendered outside <Routes> in App.js.

The code (reference)

```
import React from "react";
import AppBar from "@mui/material/AppBar";
import Toolbar from "@mui/material/Toolbar";
import Typography from "@mui/material/Typography";
import IconButton from "@mui/material/IconButton";
import ShoppingCartIcon from "@mui/icons-material/ShoppingCart";
import Badge from "@mui/material/Badge";
import { useCart } from "../context/CartContext";
import { Link } from "react-router-dom";

export default function Header() {
  const { items } = useCart();

  const totalCount = items.reduce(
    (sum, it) => sum + (it.quantity || 1),
    0
  );

  return (
    <AppBar position="sticky">
      <Toolbar>
        <Typography
          variant="h6"
          component={Link}
          to="/"
          sx={{ flexGrow: 1, textDecoration: "none", color: "inherit" }}
        >
          Ecomm Demo
        </Typography>
        <IconButton
          color="inherit"
          aria-label="cart"
          component={Link}
          to="/checkout"
        >
          <Badge badgeContent={totalCount} color="secondary">
            <ShoppingCartIcon />
          </Badge>
        </IconButton>
      </Toolbar>
    </AppBar>
  );
}
```

---

1) Imports: What each one is used for

React

import React from "react";

Needed for JSX.

Material UI components

import AppBar from "@mui/material/AppBar";

import Toolbar from "@mui/material/Toolbar";

import Typography from "@mui/material/Typography";

import IconButton from "@mui/material/IconButton";

import Badge from "@mui/material/Badge";

- **AppBar**: the top navigation bar container
- **Toolbar**: provides spacing/structure inside AppBar
- **Typography**: MUI text component with theme-friendly styles
- **IconButton**: a clickable button designed for icons
- **Badge**: small number bubble shown on top of an icon (cart count)

Material UI icon

import ShoppingCartIcon from "@mui/icons-material/ShoppingCart";

A ready-made cart icon from MUI's icon library.

Cart context hook

import { useCart } from "../context/CartContext";

This reads global cart state from CartProvider (in App.js).

React Router link

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

Used to navigate between pages **without reloading** the app.

---

2) Reading cart state with useCart()

```
const { items } = useCart();
```

- items is the cart array stored in global context.
- Each item typically looks like:

```
{
  id,
  name,
  price,
  imageUrl,
  quantity
}
```

Because Header needs to show the cart count, it must be connected to the cart state.

---

3) Computing the total item count (important logic)

```
const totalCount = items.reduce(
  (sum, it) => sum + (it.quantity || 1),
  0
);
```

What does this do?

It computes the badge number. If cart has:

- Coffee x2
- Mug x1

totalCount = 3

Why reduce?

reduce is a standard JavaScript way to sum up array values.

- sum starts at 0
- For each cart item it, add its quantity
- If quantity is missing, treat it as 1

This line is an example of “derived data”:

- We don’t store totalCount in state,
- We compute it from items when rendering.

---

4) Rendering the UI (Material UI layout)

<AppBar position="sticky">

```
<AppBar position="sticky">
```

- sticky means the header stays visible at the top while scrolling.

Other options:

- fixed (always fixed at top)
- static (normal flow, scrolls away)

<Toolbar>

```
<Toolbar>
```

Adds standard padding and aligns items horizontally.

---

5) Title as a link to Home

```
<Typography
  variant="h6"
  component={Link}
  to="/"
  sx={{ flexGrow: 1, textDecoration: "none", color: "inherit" }}
>
```

```
  Ecomm Demo
</Typography>
```

Key points

- variant="h6": typography style (theme-based heading size)
- component={Link}: renders Typography as a React Router <Link>
- to="/": clicking it navigates to home page
- sx={{ flexGrow: 1, ... }}:
    - flexGrow: 1 pushes the cart icon to the right
    - textDecoration: "none" removes underline
    - color: "inherit" keeps AppBar’s text color

This is a common MUI pattern: **turn any component into a link** by using component={Link}.

---

6) Cart icon button links to Checkout

```
<IconButton
  color="inherit"
  aria-label="cart"
  component={Link}
  to="/checkout"
>
```

Key points

- component={Link} and to="/checkout":
    - clicking the cart icon navigates to the Checkout page
- aria-label="cart":
    - accessibility feature for screen readers
- color="inherit":
    - icon uses AppBar’s text color

---

7) Badge + icon

```
<Badge badgeContent={totalCount} color="secondary">
  <ShoppingCartIcon />
</Badge>
```

- **Badge** displays totalCount as a small bubble.
- color="secondary" uses the theme's secondary color (#f50057).
- Inside Badge is the actual cart icon.

Key points:

1. **Header is a reusable component**
   - Used on every page.
2. **Context makes global state accessible anywhere**
   - Header reads cart state without props.
3. **Derived values should be computed, not stored**
   - totalCount is computed from items.
4. **React Router navigation without reload**
   - component={Link} + to="/..." is clean and fast.
5. **Material UI layout is consistent**
   - AppBar + Toolbar + Typography + IconButton + Badge

---

components/ProductCard.js (Reusable Product UI Card)

What is ProductCard.js responsible for?

ProductCard renders **one product** as a Material UI card. It:

1. Displays the product image, name, description, and price
2. Provides an **“Add to cart”** button (updates global cart state)
3. Provides a **“View”** button (navigates to the product details page)

This card is used in pages like **Home** to show a list/grid of products.

The code (reference)

```
import React from "react";
import Card from "@mui/material/Card";
import CardMedia from "@mui/material/CardMedia";
import CardContent from "@mui/material/CardContent";
import Typography from "@mui/material/Typography";
import CardActions from "@mui/material/CardActions";
import Button from "@mui/material/Button";
import { useCart } from "../context/CartContext";
import { resolveImageUrl } from "../config";
import { Link } from "react-router-dom";

export default function ProductCard({ product }) {
  const { add } = useCart();

  return (
    <Card sx={{ height: "100%", display: "flex", flexDirection: "column" }}>
      <CardMedia
        component="img"
        height="140"
        image={resolveImageUrl(product.imageUrl)}
        alt={product.name}
      />
      <CardContent sx={{ flexGrow: 1 }}>
        <Typography gutterBottom variant="h6" component="div">
          {product.name}
        </Typography>
        <Typography variant="body2" color="text.secondary">
          {product.description}
        </Typography>
        <Typography variant="subtitle1" sx={{ mt: 1 }}>
          ${product.price.toFixed(2)}
        </Typography>
      </CardContent>
      <CardActions>
        <Button size="small" onClick={() => add(product)}>
          Add to cart
        </Button>
        <Button
          size="small"
          component={Link}
          to={`/products/${product.id}`}
        >
          View
        </Button>
      </CardActions>
    </Card>
  );
}
```

---

1) Imports: what each piece is for

Material UI “Card” components

import Card from "@mui/material/Card";

import CardMedia from "@mui/material/CardMedia";

import CardContent from "@mui/material/CardContent";

import CardActions from "@mui/material/CardActions";

MUI cards have a common structure:

- **Card** → outer container
- **CardMedia** → image/video area
- **CardContent** → main text content
- **CardActions** → buttons area at the bottom

This structure improves consistency and readability.

Typography and buttons

import Typography from "@mui/material/Typography";

import Button from "@mui/material/Button";

- Typography ensures text sizes/colors match the theme.

- Button provides consistent UI buttons.

Cart context

import { useCart } from "../context/CartContext";

- Gives access to global cart actions.
- Here we use add(product) to add/increase quantity.

Image URL helper

import { resolveImageUrl } from "../config";

- Converts DB image values into a browser-friendly full URL.
- Example:
    - DB stores /images/frother.jpg
    - Browser needs http://localhost:8080/images/frother.jpg

React Router Link

import { Link } from "react-router-dom";

- Enables navigation to product details pages without reloading.

---

2) Component input: product prop

export default function ProductCard({ product }) {

This card is reusable because it receives a product object as input.

Typical product shape:

```
{
  id,
  name,
  description,
  price,
  imageUrl
}
```

This design allows pages to do:

<ProductCard product={p} />

for each product in a list.

---

3) Accessing cart action via context

```
const { add } = useCart();
```

- useCart() reads the cart context.
- add(product) either:
    - adds the product with quantity 1, or
    - increases quantity if it already exists (based on your CartContext implementation)

Key0020point:
This avoids “prop drilling”. ProductCard doesn’t need the cart function passed down manually.

---

4) Card layout and styling (sx)

```
<Card sx={{ height: "100%", display: "flex", flexDirection: "column" }}>
```

Why these styles?

- height: "100%" makes cards fill the grid cell height
- display: "flex" + flexDirection: "column" makes the card behave like a vertical layout:
    - image at top
    - content in middle
    - actions at bottom

This helps with consistent card sizes, especially in a grid layout.

---

5) Product image (CardMedia)

```
<CardMedia

  component="img"

  height="140"

  image={resolveImageUrl(product.imageUrl)}

  alt={product.name}
```

```
/>
```

Key points:

- component="img": renders as a real <img>
- height="140": fixed height for consistent card appearance
- image={resolveImageUrl(product.imageUrl)}:
  - Uses helper to build correct URL
  - Supports both:
    - relative paths (/images/coffee.jpg)
    - full URLs (CDN links)
- alt={product.name}: accessibility (screen readers) and SEO best practice

---

6) CardContent: product text and price

```
<CardContent sx={{ flexGrow: 1 }}>
```

- flexGrow: 1 makes the content area expand and push the buttons to the bottom.
- This ensures **CardActions stays aligned** across all cards.

Typography blocks

```
<Typography gutterBottom variant="h6" component="div">
  {product.name}
</Typography>
```

- variant="h6": heading style
- gutterBottom: adds bottom margin

```
<Typography variant="body2" color="text.secondary">
  {product.description}
</Typography>
```

- Smaller text (body2) and secondary color for description

```
<Typography variant="subtitle1" sx={{ mt: 1 }}>
  ${product.price.toFixed(2)}
</Typography>
```

- Shows price with 2 decimals
- mt: 1 adds spacing above

Important note:
toFixed(2) requires price to be a number. If backend sends price as string, you might need:

```
Number(product.price).toFixed(2)
```

---

7) CardActions: buttons

Add to cart button

```
<Button size="small" onClick={() => add(product)}>
  Add to cart
</Button>
```

- onClick calls add(product)
- Updates global cart state
- Header badge and Checkout page update automatically because they also read the cart context

View (go to product details)

```
<Button
  size="small"
  component={Link}
  to={`/products/${product.id}`}
>
  View
</Button>
```

- Uses component={Link} to render a router link as a button
- Navigates to dynamic route:
  - /products/5
- That route loads ProductDetail.js (defined in App.js)

Key points:

1. **Reusable component design**
    - ProductCard takes product as a prop; it doesn’t fetch data itself.
2. **Separation of concerns**
    - Page fetches data → card displays it.
3. **Global state via Context**
    - Card can add to cart without props.
4. **Routing integration**
    - Button can be turned into a link using component={Link}.
5. **Consistent UI with Material UI**
    - Card structure + sx makes layout uniform.

---

context/CartContext.js (Global Cart State with React Context)

What problem does this file solve?

In an e-commerce app, the cart must be accessible from many places:

- **Home page** → "Add to cart" button
- **Product detail page** → "Add to cart"
- **Header** → show cart badge count
- **Checkout page** → list items, change quantities

If we kept cart state inside one component, we would have to pass it down through many levels (called **prop drilling**).

React Context solves this by providing **global state** that any component can access.

The code (reference)

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

export const CartContext = createContext();

export const CartProvider = ({ children }) => {
  const [items, setItems] = useState([]);

  // Add or increment quantity
  const add = (product) => {
    setItems((prev) => {
      const existing = prev.find((it) => it.id === product.id);
      if (existing) {
        return prev.map((it) =>
          it.id === product.id
            ? { ...it, quantity: (it.quantity || 1) + 1 }
            : it
        );
      }
      return [...prev, { ...product, quantity: 1 }];
    });
  };

  // Decrease quantity by 1 (if quantity becomes 0, remove)
  const decrement = (id) => {
    setItems((prev) =>
      prev
```

```
        .map((it) =>
          it.id === id ? { ...it, quantity: (it.quantity || 1) - 1 } : it
        )
        .filter((it) => (it.quantity || 1) > 0)
    );
  };

  // Remove all of a product
  const remove = (id) => {
    setItems((prev) => prev.filter((it) => it.id !== id));
  };

  const clear = () => setItems([]);

  const value = { items, add, decrement, remove, clear };

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

// Optional convenience hook
export const useCart = () => useContext(CartContext);
```

---

1) Creating a Context

export const CartContext = createContext();

This creates a Context object.

Think of it as a **shared channel**:

- The Provider **sends** values into the channel (cart data + functions).
- Any component can **read** the values from that channel.

---

2) CartProvider: the component that holds the cart state

export const CartProvider = ({ children }) => {

Why children?

CartProvider wraps part of the component tree:

<CartProvider>

  <App />

```
</CartProvider>
```

So children means: “Render whatever is inside the provider.”

Cart state

```
const [items, setItems] = useState([]);
```

- items is the cart array.
- Starts empty: []
- Each element is a product with a quantity, for example:

```
{
  id: 5,
  name: "Milk Frother",
  price: 24.99,
  imageUrl: "/images/frother.jpg",
  quantity: 2
}
```

---

3) The add(product) function (add or increase quantity)

```
const add = (product) => {
  setItems((prev) => {
    const existing = prev.find((it) => it.id === product.id);
    if (existing) {
      return prev.map((it) =>
        it.id === product.id
          ? { ...it, quantity: (it.quantity || 1) + 1 }
          : it
      );
    }
    return [...prev, { ...product, quantity: 1 }];
  });
};
```

Key idea:

- If product is already in cart → **increment quantity**
- Else → add it with quantity = 1

Important technique: functional state update

setItems((prev) => { ... })

This is best practice because it uses the **latest** state value, even if multiple updates happen quickly.

Important technique: immutability

Notice we never modify prev directly.

- We create a new array using:
    - map(...) or [...prev, newItem]
- This is important because React detects changes by reference.

---

4) decrement(id) (reduce quantity, remove if it reaches 0)

```
const decrement = (id) => {
  setItems((prev) =>
    prev
      .map((it) =>
        it.id === id ? { ...it, quantity: (it.quantity || 1) - 1 } : it
      )
      .filter((it) => (it.quantity || 1) > 0)
  );
};
```

What it does:

1. It decreases the quantity by 1 for the selected product
2. If quantity becomes 0, it removes it from the cart

This is done in two steps:

- map(...) → update quantity
- filter(...) → remove items whose quantity is now 0

---

5) remove(id) (remove product completely)

```
const remove = (id) => {
  setItems((prev) => prev.filter((it) => it.id !== id));
};
```

This removes the product entirely (regardless of quantity).

---

6) clear() (empty the cart)

```
const clear = () => setItems([]);
```

Simple reset back to an empty array.

---

7) Provider value: what we expose to the rest of the app

```
const value = { items, add, decrement, remove, clear };
```

This is what other components get access to through the context.

Example:

- Header uses items to compute badge count
- Checkout uses items, decrement, remove, clear
- ProductCard uses add

---

8) Providing the value to children

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

This line “publishes” the cart state to every component inside the provider.

---

9) Convenience hook: useCart()

```
export const useCart = () => useContext(CartContext);
```

Instead of writing:

```
const cart = useContext(CartContext);
```

every time, you write:

```
const { items, add } = useCart();
```

Cleaner and more readable.

Key points:

1. **Context is for shared state across many components**
   - Avoids prop drilling.
2. **Provider holds the state**
   - The Provider wraps the app, and all children can access the state.
3. **Immutability is essential**
   - Use map, filter, and spread ... to create new arrays/objects.
4. **Functional updates (setState(prev => ...)) are best practice**
   - Prevents bugs when multiple updates happen.
5. **Actions (add/decrement/remove/clear) represent cart operations**
   - This resembles a simple “store” pattern.

---

pages/ProductDetail.js (Dynamic Product Detail Page)

What does this page do?

ProductDetail.js is the page shown when the user visits a URL like:

- /products/1
- /products/5

It:

1. Reads the product **id** from the URL
2. Calls the backend API to fetch that product's data
3. Shows:
    - loading spinner while fetching
    - error message if something fails
    - product details and large image if successful
4. Provides buttons:
    - **Add to cart**
    - **Back to Home**

Code:

```
// src/pages/ProductDetail.js
import React, { useEffect, useState } from "react";
import { useParams, Link } from "react-router-dom";
import { getProductById } from "../services/productService";
import { useCart } from "../context/CartContext";
import { resolveImageUrl } from "../config";

import Container from "@mui/material/Container";
import Grid from "@mui/material/Grid";
import Typography from "@mui/material/Typography";
import Button from "@mui/material/Button";
import Box from "@mui/material/Box";
import CircularProgress from "@mui/material/CircularProgress";
import CardMedia from "@mui/material/CardMedia";

export default function ProductDetail() {
  const { id } = useParams();
  const { add } = useCart();

  const [product, setProduct] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    let cancelled = false;

    setLoading(true);
    setError(null);

    getProductById(id)
      .then((data) => {
        if (!cancelled) {
          console.log("Loaded product", data);
          setProduct(data);
          setLoading(false);
        }
      })
      .catch((err) => {
        if (!cancelled) {
          console.error("Failed to load product", err);
          setError(err?.message || "Error loading product");
          setLoading(false);
        }
      });

    return () => {
      cancelled = true;
    };
  }, [id]);

  if (loading) {
    return (
      <Container sx={{ py: 4, display: "flex", justifyContent: "center" }}>
        <CircularProgress />
      </Container>
    );
  }

  if (error || !product) {
    return (
      <Container sx={{ py: 4 }}>
        <Typography variant="h5" color="error" gutterBottom>
          Could not load product.
        </Typography>
        {error && (
          <Typography variant="body2" color="text.secondary" paragraph>
            {String(error)}
          </Typography>
        )}
```

```
        <Button variant="contained" component={Link} to="/">
          Back to Home
        </Button>
      </Container>
    );
  }

  const priceText =
    typeof product.price === "number"
      ? product.price.toFixed(2)
      : product.price;

  return (
    <Container sx={{ py: 4 }}>
      <Grid container spacing={4}>
        {/* Sol: büyük resim */}
        <Grid item xs={12} md={5}>
          <CardMedia
            component="img"
            image={resolveImageUrl(product.imageUrl)}
            alt={product.name}
            sx={{
              width: "100%",
              maxHeight: 400,
              objectFit: "contain",
              borderRadius: 2,
              boxShadow: 1,
            }}
          />
        </Grid>

        {/* Sağ: bilgiler */}
        <Grid item xs={12} md={7}>
          <Typography variant="h4" gutterBottom>
            {product.name}
          </Typography>

          <Typography variant="h6" color="primary" gutterBottom>
            ${priceText}
          </Typography>

          {product.description && (
            <Typography variant="body1" paragraph>
              {product.description}
            </Typography>
          )}

          {product.details && (
            <Typography variant="body2" color="text.secondary" paragraph>
```

```
              {product.details}
            </Typography>
          )}

          <Box sx={{ mt: 3, display: "flex", gap: 2, flexWrap: "wrap" }}>
            <Button
              variant="contained"
              size="large"
              onClick={() => add(product)}
            >
              Add to cart
            </Button>
            <Button
              variant="outlined"
              size="large"
              component={Link}
              to="/"
            >
              Back to Home
            </Button>
          </Box>
        </Grid>
      </Grid>
    </Container>
  );
}
```

---

1) Imports: what each group is for

React hooks

import React, { useEffect, useState } from "react";

- useState stores page state: product, loading, error
- useEffect runs code when the page loads or when id changes

Routing helpers

import { useParams, Link } from "react-router-dom";

- useParams() reads :id from the route /products/:id
- Link allows navigation without refreshing the page

Service + cart + image helper

import { getProductById } from "../services/productService";

import { useCart } from "../context/CartContext";

import { resolveImageUrl } from "../config";

- getProductById(id) calls the backend
- useCart() provides the add(product) function
- resolveImageUrl converts /images/x.jpg → http://localhost:8080/images/x.jpg

Material UI components

import Container from "@mui/material/Container";

import Grid from "@mui/material/Grid";

import Typography from "@mui/material/Typography";

import Button from "@mui/material/Button";

import Box from "@mui/material/Box";

import CircularProgress from "@mui/material/CircularProgress";

import CardMedia from "@mui/material/CardMedia";

- Provide consistent layout and UI styling using the theme.

---

2) Reading the product id from the URL

const { id } = useParams();

If the URL is:

- /products/5

then:

- id becomes "5" (note: it's a string)

This id is used to call:

- GET /api/products/5

---

3) Accessing cart action

const { add } = useCart();

This gives the page a function to update global cart state:

- add(product) adds the current product to the cart or increments quantity

Because the cart is in Context, adding here updates:

- Header badge count
- Checkout page list

automatically.

---

4) Page state: product, loading, error

```
const [product, setProduct] = useState(null);
const [loading, setLoading] = useState(true);
const [error, setError] = useState(null);
```

This is a standard pattern for API pages:

- product holds the fetched data
- loading controls the spinner
- error shows a message if request fails

---

5) Fetching the product with useEffect

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

  setLoading(true);
  setError(null);

  getProductById(id)
    .then((data) => {
      if (!cancelled) {
        console.log("Loaded product", data);
        setProduct(data);
        setLoading(false);
      }
```

```
   })
   .catch((err) => {
     if (!cancelled) {
       console.error("Failed to load product", err);
       setError(err?.message || "Error loading product");
       setLoading(false);
     }
   });

  return () => {
    cancelled = true;
  };
}, [id]);
```

What triggers this effect?

[id] dependency means:

- Run on first render
- Run again if the URL changes from /products/5 to /products/6

This allows users to switch products without reloading the app.

Why do we reset loading and error?

```
setLoading(true);
setError(null);
```

So the UI shows a spinner and clears old errors whenever a new product id loads.

Why the cancelled flag?

This prevents a common React problem:

- User navigates away quickly
- The request finishes after the component unmounted
- Then React would warn:

“Can’t perform a React state update on an unmounted component”

So we do:

- cancelled = true in cleanup
- check if (!cancelled) before setState(...)

This is a safe pattern for async requests.

---

6) Conditional rendering: loading state

```
if (loading) {
  return (
    <Container sx={{ py: 4, display: "flex", justifyContent: "center" }}>
      <CircularProgress />
    </Container>
  );
}
```

If data is still being fetched, we show a centered spinner.

Key point:
Return early for loading/error states — it keeps UI logic clean.

---

7) Conditional rendering: error state

```
if (error || !product) {
  return (
    <Container sx={{ py: 4 }}>
      <Typography variant="h5" color="error" gutterBottom>
        Could not load product.
      </Typography>
      {error && (
        <Typography variant="body2" color="text.secondary" paragraph>
          {String(error)}
        </Typography>
      )}
      <Button variant="contained" component={Link} to="/">
        Back to Home
      </Button>
    </Container>
  );
}
```

If request failed or product is missing:

- Show error message
- Show error text (if available)
- Provide a “Back to Home” button

This improves user experience.

---

8) Handling price formatting safely

```
const priceText =
  typeof product.price === "number"
    ? product.price.toFixed(2)
    : product.price;
```

Why?

- Sometimes backend returns price as a number
- Sometimes it may come as a string (depending on serialization)

toFixed(2) works only on numbers, so this prevents runtime errors.

---

9) UI layout: Grid with image on left, text on right

```
<Container sx={{ py: 4 }}>
  <Grid container spacing={4}>
```

- Container centers the content and adds padding.
- Grid container creates a responsive layout.

Left side: big image

```
<Grid item xs={12} md={5}>
  <CardMedia
    component="img"
    image={resolveImageUrl(product.imageUrl)}
    alt={product.name}
    sx={{
      width: "100%",
      maxHeight: 400,
      objectFit: "contain",
      borderRadius: 2,
      boxShadow: 1,
    }}
  />
</Grid>
```

Key points:

- xs={12} means on mobile, take full width
- md={5} means on medium+ screens, take 5/12 width
- resolveImageUrl ensures correct backend image URL
- objectFit: "contain" keeps the full image visible without cropping

Right side: product information

```
<Grid item xs={12} md={7}>
```

- On mobile: full width
- On desktop: 7/12 width

---

10) Displaying text fields conditionally

```
{product.description && (
  <Typography variant="body1" paragraph>
```

```
    {product.description}
  </Typography>
)}
```

This pattern means:

- Only show description if it exists (not null/empty)

Same for details:

```
{product.details && (
  <Typography variant="body2" color="text.secondary" paragraph>
    {product.details}
  </Typography>
)}
```

---

11) Action buttons: Add to cart + Back to Home

```
<Box sx={{ mt: 3, display: "flex", gap: 2, flexWrap: "wrap" }}>
  <Button
    variant="contained"
    size="large"
    onClick={() => add(product)}
  >
    Add to cart
  </Button>
  <Button
    variant="outlined"
    size="large"
    component={Link}
    to="/"
  >
    Back to Home
  </Button>
</Box>
```

- Box is a layout helper
- display: flex arranges buttons horizontally
- gap: 2 adds spacing between them
- flexWrap: wrap prevents overflow on small screens

Buttons:

- **Add to cart** calls context function add(product)
- **Back to Home** uses Router Link navigation

Key points

1. **Dynamic routing**
   - /products/:id + useParams()
2. **Service layer usage**
   - Page calls getProductById, not Axios directly
3. **Three-state UI for API pages**
   - loading, error, success
4. **Safe async handling**
   - cancelled flag prevents “state update after unmount”
5. **Material UI responsive layout**
   - Grid xs={12} md={...}
6. **Global cart state via Context**
   - useCart().add(product)

---

pages/Checkout.js (Cart List + Quantity Controls + Total)

What does this page do?

The Checkout page displays everything in the shopping cart:

1. If the cart is empty → show an "empty cart" message and a button back to Home.
2. If the cart has items → show a list of products with:
   - image
   - name + unit price
   - quantity controls (+ / −)
   - line total (unit price × quantity)
   - delete button
3. Show the cart **total price** and a **Proceed** button (payment not implemented).

Code:

```
import React from "react";
import { useCart } from "../context/CartContext";
import Container from "@mui/material/Container";
import Typography from "@mui/material/Typography";
import Box from "@mui/material/Box";
import Grid from "@mui/material/Grid";
import Button from "@mui/material/Button";
import Divider from "@mui/material/Divider";
import CardMedia from "@mui/material/CardMedia";
import IconButton from "@mui/material/IconButton";
import AddIcon from "@mui/icons-material/Add";
import RemoveIcon from "@mui/icons-material/Remove";
import DeleteIcon from "@mui/icons-material/Delete";
import { resolveImageUrl } from "../config";
import { Link } from "react-router-dom";

export default function Checkout() {
  const { items, add, decrement, remove, clear } = useCart();

  const totalPrice = items.reduce(
    (sum, it) => sum + it.price * (it.quantity || 1),
    0
  );

  if (items.length === 0) {
    return (
      <Container sx={{ py: 4 }}>
```

```
        <Typography variant="h4" gutterBottom>
          Your cart is empty
        </Typography>
        <Button variant="contained" component={Link} to="/">
          Go back to shopping
        </Button>
      </Container>
    );
  }

  return (
    <Container sx={{ py: 4 }}>
      <Typography variant="h4" gutterBottom>
        Checkout
      </Typography>

      <Box sx={{ mb: 2 }}>
        <Button color="secondary" onClick={clear}>
          Clear cart
        </Button>
      </Box>

      <Divider sx={{ mb: 2 }} />

      {items.map((item) => (
        <Box key={item.id} sx={{ mb: 2 }}>
          <Grid container alignItems="center" spacing={2}>
            <Grid item xs={12} sm={2}>
              <CardMedia
                component="img"
                image={resolveImageUrl(item.imageUrl)}
                alt={item.name}
                sx={{ height: 80, width: "auto", borderRadius: 1 }}
              />
            </Grid>
            <Grid item xs={12} sm={4}>
              <Typography variant="subtitle1">{item.name}</Typography>
              <Typography variant="body2" color="text.secondary">
                ${item.price.toFixed(2)} each
              </Typography>
            </Grid>
            <Grid
              item
              xs={12}
              sm={3}
              sx={{ display: "flex", alignItems: "center", gap: 1 }}
            >
              <IconButton onClick={() => decrement(item.id)}>
                <RemoveIcon />
              </IconButton>
              <Typography>{item.quantity || 1}</Typography>
              <IconButton onClick={() => add(item)}>
                <AddIcon />
              </IconButton>
            </Grid>
            <Grid item xs={8} sm={2}>
              <Typography variant="subtitle1">
                ${(item.price * (item.quantity || 1)).toFixed(2)}
              </Typography>
            </Grid>
            <Grid item xs={4} sm={1}>
              <IconButton onClick={() => remove(item.id)}>
                <DeleteIcon />
              </IconButton>
            </Grid>
          </Grid>
          <Divider sx={{ mt: 2 }} />
        </Box>
      ))}

      <Box
        sx={{
          mt: 3,
          display: "flex",
          justifyContent: "space-between",
          alignItems: "center",
          flexWrap: "wrap",
          gap: 2,
        }}
      >
        <Typography variant="h5">
          Total: ${totalPrice.toFixed(2)}
        </Typography>
        <Button
          variant="contained"
          size="large"
          onClick={() => {
            alert("Proceeding to payment… (not implemented)");
          }}
        >
          Proceed
        </Button>
      </Box>
    </Container>
  );
}
```

---

1) Imports: what each group is for

Cart context

import { useCart } from "../context/CartContext";

Gives access to cart data (items) and actions (add, decrement, remove, clear).

Material UI layout & UI components

import Container from "@mui/material/Container";

import Typography from "@mui/material/Typography";

import Box from "@mui/material/Box";

import Grid from "@mui/material/Grid";

import Button from "@mui/material/Button";

import Divider from "@mui/material/Divider";

import CardMedia from "@mui/material/CardMedia";

import IconButton from "@mui/material/IconButton";

- **Container**: centers content and adds padding
- **Typography**: consistent text styles
- **Box**: flexible layout helper (CSS via sx)
- **Grid**: responsive row/column layout
- **Divider**: horizontal line separators
- **CardMedia**: displays images
- **IconButton**: compact buttons for icons

Icons

import AddIcon from "@mui/icons-material/Add";

import RemoveIcon from "@mui/icons-material/Remove";

import DeleteIcon from "@mui/icons-material/Delete";

Used for +, −, and delete actions.

Image helper + routing

import { resolveImageUrl } from "../config";

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

- resolveImageUrl builds full URL for backend images.
- Link allows navigation back to Home without page reload.

---

2) Reading cart state and actions from context

```
const { items, add, decrement, remove, clear } = useCart();
```

This page needs both **data** and **actions**:

- items → array of cart items
- add(item) → increase quantity by 1 (reuses the same cart logic)
- decrement(id) → decrease quantity by 1, remove if becomes 0
- remove(id) → remove completely
- clear() → empty the cart

---

3) Computing the total price (derived data)

```
const totalPrice = items.reduce(
  (sum, it) => sum + it.price * (it.quantity || 1),
  0
);
```

What it does

Adds up the cart total:

- For each item:
  - lineTotal = price × quantity
- Sum all line totals.

Why (it.quantity || 1)?

If quantity is missing (undefined), treat it as 1 to avoid NaN.

Key point:
Total price is **not stored** in state; it is computed from items during rendering.

---

4) Empty cart UI (conditional rendering)

```
if (items.length === 0) {
  return (
    <Container sx={{ py: 4 }}>
      <Typography variant="h4" gutterBottom>
        Your cart is empty
      </Typography>
      <Button variant="contained" component={Link} to="/">
        Go back to shopping
      </Button>
    </Container>
  );
}
```

This is a common pattern:

- Early return for “empty state”
- Improves UX and keeps logic clean

---

5) Main layout when cart has items

Page title

```
<Typography variant="h4" gutterBottom>
  Checkout
</Typography>
```

Clear cart button

```
<Button color="secondary" onClick={clear}>
  Clear cart
</Button>
```

- Calls clear() from context
- Immediately empties cart

- Header badge updates automatically

Divider

```
<Divider sx={{ mb: 2 }} />
```

Separates header controls from item list.

---

6) Rendering cart items with map()

```
{items.map((item) => (
  <Box key={item.id} sx={{ mb: 2 }}>
    ...
  </Box>
))}
```

Key teaching points

- map() creates one UI block per cart item.
- key={item.id} is required so React can efficiently update lists.

---

7) Each cart row layout (Grid)

```
<Grid container alignItems="center" spacing={2}>
```

Grid container makes a responsive row.

Column 1: product image

```
<Grid item xs={12} sm={2}>
  <CardMedia
    component="img"
    image={resolveImageUrl(item.imageUrl)}
    alt={item.name}
    sx={{ height: 80, width: "auto", borderRadius: 1 }}
  />
</Grid>
```

- xs={12} → full width on mobile

- sm={2} → small screens and above: 2/12 width (small column)
- Uses backend image URL helper.

Column 2: name + unit price

```
<Grid item xs={12} sm={4}>
  <Typography variant="subtitle1">{item.name}</Typography>
  <Typography variant="body2" color="text.secondary">
    ${item.price.toFixed(2)} each
  </Typography>
</Grid>
```

- Shows name and price per item.
- toFixed(2) formats currency nicely.

Column 3: quantity controls (− quantity +)

```
<Grid item xs={12} sm={3} sx={{ display: "flex", alignItems: "center", gap: 1 }}>
  <IconButton onClick={() => decrement(item.id)}>
    <RemoveIcon />
  </IconButton>
  <Typography>{item.quantity || 1}</Typography>
  <IconButton onClick={() => add(item)}>
    <AddIcon />
  </IconButton>
</Grid>
```

Actions:

- **RemoveIcon** → decreases quantity
- **AddIcon** → increases quantity
- Display quantity in the middle

Nice design choice:
Checkout page reuses the same cart logic:

- increment = add(item)
- decrement = decrement(item.id)

No duplicate quantity logic in the page.

Column 4: line total (price × quantity)

```
<Grid item xs={8} sm={2}>
  <Typography variant="subtitle1">
    ${(item.price * (item.quantity || 1)).toFixed(2)}
  </Typography>
</Grid>
```

- On mobile (xs={8}) uses more space.
- Shows total cost for that product row.

Column 5: delete icon (remove completely)

```
<Grid item xs={4} sm={1}>
  <IconButton onClick={() => remove(item.id)}>
    <DeleteIcon />
  </IconButton>
</Grid>
```

- Deletes the item from cart entirely.

Divider after each item row

```
<Divider sx={{ mt: 2 }} />
```

Improves readability.

---

8) Footer area: total + proceed button

```
<Box sx={{ mt: 3, display: "flex", justifyContent: "space-between", ... }}>
```

This creates a responsive footer section:

- total price on the left
- proceed button on the right
- wraps nicely on small screens

Total

```
<Typography variant="h5">
  Total: ${totalPrice.toFixed(2)}
</Typography>
```

Proceed button (placeholder)

```
<Button
  variant="contained"
  size="large"
  onClick={() => {
    alert("Proceeding to payment… (not implemented)");
  }}
>
  Proceed
</Button>
```

This is intentional for the course project:

- Demonstrates the flow
- Does not implement payment logic
- Shows where the next step would go

Key points:

1. **Global state usage**
   - Checkout reads cart state and updates it using context actions.
2. **Derived data**

- Total price is computed with reduce, not stored.

3. **List rendering**
    - items.map + key={item.id} for dynamic lists.
4. **Responsive layout**
    - Grid xs={12} sm={...} adapts to mobile/desktop.
5. **Reusable logic**
    - Quantity increments reuse add(item) from the cart context.
6. **Good UX patterns**
    - Empty state screen
    - Clear cart button
    - Line totals and final total

---

pages/Home.js (Product List Page)

What does this page do?

Home.js is the main landing page of the e-commerce site. It:

1. Fetches products from the backend when the page loads
2. Shows a **Loading...** message while the request is in progress
3. Displays products in a responsive **Material UI Grid**
4. Uses a reusable component (ProductCard) to render each product

---

The code (reference)

```
import React, { useEffect, useState } from 'react';
import Container from '@mui/material/Container';
import Grid from '@mui/material/Grid';
import Typography from '@mui/material/Typography';
import ProductCard from '../components/ProductCard';
import { getAllProducts } from '../services/productService';
//import { getTopProducts } from '../services/productService';

export default function Home() {
  const [products, setProducts] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    let mounted = true;
    getAllProducts()
      .then((data) => {
        if (mounted) {
          setProducts(data);
          setLoading(false);
        }
      })
      .catch(() => {
        if (mounted) setLoading(false);
      });

    return () => {
      mounted = false;
    };
  }, []);

  return (
    <Container sx={{ py: 3 }}>
      <Typography variant="h4" component="h1" gutterBottom>
```

```
        Featured Products
      </Typography>
      {loading ? (
        <Typography>Loading...</Typography>
      ) : (
        <Grid container spacing={2} alignItems="stretch">
          {products.map((p) => (
            <Grid item key={p.id} xs={12} sm={6} md={4}>
              <ProductCard product={p} />
            </Grid>
          ))}
        </Grid>
      )}
    </Container>
  );
}
```

---

1) Imports: what each one is for

React hooks

import React, { useEffect, useState } from 'react';

- useState stores component state (products, loading)
- useEffect runs side effects such as API calls after the component renders

Material UI layout

import Container from '@mui/material/Container';

import Grid from '@mui/material/Grid';

import Typography from '@mui/material/Typography';

- **Container**: centers content and provides page padding
- **Grid**: responsive layout for product cards
- **Typography**: consistent text styles

Reusable component

import ProductCard from '../components/ProductCard';

- Displays one product as a card (image, name, price, buttons)

API function (service layer)

import { getAllProducts } from '../services/productService';

- Calls the backend:
    - GET /api/products
- Returns a list/array of products.

Note: getTopProducts is commented because the code switched from “random top” to “list all/featured”.

---

2) State: storing products and loading flag

```
const [products, setProducts] = useState([]);

const [loading, setLoading] = useState(true);
```

- products: holds the list of products returned from the backend
- loading: controls whether to show “Loading...” or show the grid

Typical pattern:

- Start with loading = true
- After request finishes (success or failure), set loading = false

---

3) Fetching data with useEffect

```
useEffect(() => {
 let mounted = true;


 getAllProducts()
  .then((data) => {
   if (mounted) {
    setProducts(data);
    setLoading(false);
   }
  })
  .catch(() => {
   if (mounted) setLoading(false);
```

```
    });


  return () => {
    mounted = false;
  };
}, []);
```

When does this run?

The dependency array is empty:

```
}, []);
```

So the effect runs:

- only **once** when the Home page first loads (component mounts)

Why do we use mounted?

This is a safety pattern to avoid updating state after the component unmounts.

Scenario:

- User enters Home page
- The API request starts
- Before the request finishes, user navigates away
- If the request finishes later, setProducts would try to run on an unmounted component → warnings in React

So:

- mounted = true initially
- On cleanup (unmount), set mounted = false
- Only update state if mounted is still true

(Equivalent to the cancelled pattern you saw in ProductDetail.)

Error handling

```
.catch(() => {
  if (mounted) setLoading(false);
});
```

If the request fails:

- we still stop the loading state
- but we do not show an error message here (simple version)
    - In a more advanced version you could add error state and show a message.

---

4) Rendering the UI

Page container

```
<Container sx={{ py: 3 }}>
```

- Adds vertical padding (py: 3)
- Keeps content centered and readable

Page title

```
<Typography variant="h4" component="h1" gutterBottom>
  Featured Products
</Typography>
```

- variant="h4": large heading style
- component="h1": semantically a main heading
- gutterBottom: adds spacing below

---

5) Conditional rendering: loading vs content

```
{loading ? (
  <Typography>Loading...</Typography>
) : (
  ...
)}
```

If loading is true:

- show “Loading...”

Otherwise:

- show product grid

This is a very common React pattern.

---

6) Rendering products in a responsive Grid

```
<Grid container spacing={2} alignItems="stretch">
  {products.map((p) => (
    <Grid item key={p.id} xs={12} sm={6} md={4}>
      <ProductCard product={p} />
    </Grid>
  ))}
</Grid>
```

Grid container

- spacing={2}: spacing between cards
- alignItems="stretch": makes grid items stretch to equal height (helps equal-sized cards)

Grid item sizing

xs={12} sm={6} md={4}

This means:

- **mobile (xs)**: 12/12 width → 1 card per row
- **small screens (sm)**: 6/12 width → 2 cards per row
- **medium+ (md)**: 4/12 width → 3 cards per row

So the UI adapts nicely across devices.

Mapping products to cards

- products.map(...) creates one ProductCard per product
- key={p.id} is required for React list performance and correctness

---

Key points:

1. **Service layer use**
    - Home does not use axios directly; it calls getAllProducts().

2. **useEffect for API calls**
    - Fetch data on mount using useEffect(..., []).
3. **State for data + loading**
    - products state holds backend data.
    - loading state controls UI feedback.
4. **Conditional rendering**
    - Show loading message while waiting.
5. **Reusable UI components**
    - Home only arranges the grid; ProductCard renders each product.
6. **Responsive layout**
    - Grid xs={12} sm={6} md={4} is a standard responsive pattern.

The End