Project : Simple Shopping Cart

Project Overview

- The goal of this project is to create a simple text-based shopping cart system.

Project Description

- **Task :** Create a program that simulates a shopping cart where users can add items, view the cart, remove items, and view the total price.
- **Items :** The available items are stored in a list of tuples, where each tuple contains the item name and its price.
- **Cart :** The cart is a list where users can add or remove items.

Steps

1. **Create a List of Items :** Create a list of tuples where each tuple contains an item name and its price.
2. **Menu Options :** Display a menu with the following options :
   - **A.** View available items.
   - **B.** Add item to cart.
   - **C.** View cart.
   - **D.** Remove item from cart.
   - **E.** View total price.
   - **F.** Exit.
3. **Implement Each Menu Option :**
   - **A.** **View Available Items :** Display all available items with their prices.
   - **B.** **Add Item to Cart :** Allow the user to select an item to add to the cart.
   - **C.** **View Cart :** Display all items currently in the cart.

D. **Remove Item from Cart :** Allow the user to remove an item from the cart.

E. **View Total Price :** Calculate and display the total price of all items in the cart.

F. **Exit :** End the program.

Frontend Stack

1. **Framework :** React + Vite (fast dev. Refresh)
2. **State & Data :** React Context for cart state + useReducer & Zustand
3. **Routing :** React router v6 - nested routes for checkout steps
4. **Forms & Validations :** React Hook Form + Yup schemes
5. **Styling :** Tailwind CSS (quick responsive grids)
6. **Icons & Illustrations :** lucide - react icons

1. **config/env.ts** -> purpose -> Uses dotenv to load and *validate* env vars. Exports a typed env object. Exports the typed `env` object with `PORT`, `NODE_ENV`, and optionally `SSL_KEY_PATH`, `SSL_CERT_PATH`. It loads variables from `.env`, validates them with **zod**, and exports a frozen, strongly-typed `env` object the rest of your codebase can trust.

2. **config/logger.ts** -> purpose -> pino/winston instance with a .stream so morgan can pipe access logs. Gives us the JSON-structured logs in production (perfect for CloudWatch, Datadog, Logstash, …)

Gives us the Colorised, human-readable logs in development via **`pino-pretty`**

A `loggerStream` helper so `morgan` can pipe HTTP access logs straight into the same sink

3. **utils/apiError.ts** -> purpose -> Tiny class that extends Error and stores statusCode (e.g. NotFoundError)

4. **middlewares/error.middleware.ts** -> purpose -> Centralised Express error middleware that formats the API response and logs stack traces.

5. **feature *.routes.ts files** -> purpose -> Each exports an `express.Router()` with its feature endpoints.

6. **config/db.ts** -> purpose -> Exports a *single* Prisma client instance: export const prisma = new PrismaClient(). It keeps just **one** connection alive—even during hot-reload in dev—and exposes a typed client you can import anywhere. P**risma singleton** for src/config/db.ts

Prisma Singleton Pattern -> Prevents “Too many connections” when ts-node-dev / nodemon reload. Attaches the client to globalThis in DEV; production just exports.

Singleton via `globalThis` `->` Avoids multiple DB handles when Vite / ts-node-dev restarts modules.

7. `zod` schema -> Catches typos & invalid values *at startup*, not at runtime.

8. `PORT` coercion -> accepts `"3000"` (string) or nothing and always outputs a **number**.

9. config/passport.ts -> Registers a Passport-JWT strategy -> Accepts the token from either:
1. Authorization header → "Bearer <token>”

2. Signed cookie → token=<token> , When valid, attaches the full User record to req.user

1. **auth.middleware.ts** -> Verifies the incoming JWT (via Passport-JWT). If valid, \`req.user\` is populated with the full User record. If missing / invalid, responds with 401.
2. **jwtAuth** -> Wraps \`passport.authenticate('jwt')\` so we can keep the same middleware signature (\`(req, res, next)\`) everywhere.
3. **error.middleware.ts** -> Global Express error-handler. Converts every thrown/forwarded error into a consistent JSON envelope. Recognises: – ApiError (our custom class → status + message) – ZodError (validation failures → 400 + Issues[]) – Anything else (fallback → 500). Logs the full stack with pino/winston.
4. **validate.middleware.ts** -> Tiny Zod-powered validator for Express routes. Pass an object containing Zod schemas for any of \`body\`, \`query\`, \`params\`. If validation succeeds, the parsed (and therefore typed & sanitized) data, overwrites \`req.body\`, \`req.query\`, or \`req.params\`. If it fails, we forward a 400 ApiError so the global error handler responds, with a uniform JSON envelope.
5. **auth.controller.ts** -> High-level Auth endpoints: POST /api/auth/register – create account. POST /api/auth/login – issue JWT cookie. GET /api/auth/me – current user profile. POST /api/auth/logout – clear cookie.
6. **auth.service.ts** -> Encapsulates all authentication business-logic so controllers stay thin.
7. **auth.routes.ts** -> Route layer for everything under /api/auth. POST /register – create account. POST /login – issue JWT cookie. GET /me – current user (protected). POST /logout – clear cookie.

8. **user.model.ts** -> Thin, type-safe wrapper around the Prisma `user` table. Keeps all low-level DB calls for the **User** domain in one place so, services/controllers can stay clean and mock this layer in tests.
9. **user.service.ts** -> Business-logic layer for the **User** domain. Keeps controllers thin and makes unit-testing a breeze.
10. **product.model.ts** -> Thin Prisma wrapper for the **Product** domain. Centralises all DB logic (queries, filters, pagination). Keeps services/controllers clean and mock-friendly.
11. **product.service.ts** -> Business-logic for the **Product** domain. It handles: • Public catalog listing with filters + pagination. • Single-product retrieval by id OR slug. • Admin-facing CRUD helpers (create / update / delete).
12. **product.controller.ts** -> HTTP handlers for everything under /api/products
13. **cartItem.model.ts** -> Prisma wrapper for the **Cart** domain. Keeps all cart-related DB operations in one place. Makes service / controller layers cleaner and easier to unit-test.
14. **cart.service.ts** -> Business-logic for the **Cart** domain. Functions exposed: • getUserCart • addItemToCart • updateCartItemQty. • removeItemFromCart. • clearUserCart.
15. **cart.controller.ts** -> HTTP handlers for everything under /api/cart.
16. **order.model.ts** -> Prisma wrapper for the **Order** domain. Consolidates all DB calls so services/controllers are thin & testable. Exposes a reusable `select` shape for safe public-facing fields.
17. **orderItem.model.ts** -> Convenience wrapper around the `orderItem` table. While most flows create order-items *through* `OrderModel.create`, Having a stand-alone model is handy for: bulk inserts during checkout. admin reports (e.g. top-selling products).
18. **order.service.ts** -> Business-logic for the **Order** domain (a.k.a. “checkout”).
19. **category.model.ts** -> Lightweight Prisma wrapper for the **Category** domain. Only a handful of helpers are provided because categories are fairly static. Having this file keeps future refactors (e.g., caching) in one place.
20. **apiResponse.ts** -> A tiny utility that guarantees every controller returns the same envelope shape.
21. **pagination.ts** -> Tiny helpers that transform “page / limit” inputs (1-based) into the Prisma-friendly “skip / take” pair and back into a useful response meta.
22. **slugify.ts** -> A tiny, dependency-free helper that converts arbitrary text into a URL-friendly “slug” (kebab-case).
23. **express.d.ts** -> Global type-augmentation so TypeScript knows that `req.user` exists and carries our chosen user fields. Works seamlessly with Passport.
24.