Full-Stack E-Commerce App — Step-by-Step Exercise

Welcome

In this exercise you will build a **full-stack e-commerce application** from scratch. By the end, you'll have a working online shop with user authentication, product browsing, and a shopping cart — just like a real-world app.

Don't worry if this feels big. We'll build it piece by piece, and each step connects to the one before it. Read the **"Why?"** sections — they explain how each piece fits into the bigger picture.

Feature Requirements

#	Feature	Description
1	Register & Login	Users can sign up and log in. Authentication is handled with JWT tokens.
2	Logout	Users can log out (token is removed client-side).
3	Products Page	A main page showing all available products (visible to everyone).
4	Single Product Page	Clicking a product shows its details with an "Add to Cart" button.
5	Auth Guard on Cart	If a guest clicks "Add to Cart", they are redirected to the login page.
6	Shopping Cart	Logged-in users can view their cart, change quantities, and remove items.
7	Place Order	Logged-in users can place an order from their cart.

Database Design

MongoDB with the native `mongodb` driver — NO Mongoose.

You write your own queries with `db.collection(...).find()`, etc.

Collection: `users`

```
{
  "_id": "ObjectId",
  "name": "John Doe",
  "email": "john@example.com",
  "password": "$2b$10$hashedPasswordHere"
}
```

Collection: `products`

```
{
  "_id": "ObjectId",
  "name": "Wireless Headphones",
  "description": "High quality noise-cancelling headphones",
  "price": 59.99,
  "image": "https://via.placeholder.com/300",
  "category": "electronics",
  "stock": 25
}
```

Collection: `userProducts` (the shopping cart)

```
{
  "_id": "ObjectId",
  "userId": "ObjectId",
  "productId": "ObjectId",
  "quantity": 2
}
```

Collection: `orders`

```
{
  "_id": "ObjectId",
  "userId": "ObjectId",
  "items": [
    { "productId": "ObjectId", "name": "Wireless Headphones", "price":
59.99, "quantity": 2 }
  ],
  "total": 119.98,
  "createdAt": "2025-01-15T10:30:00Z"
}
```

Folder Structure

Study these structures **before you start coding**. They represent the architecture you'll follow.

Server (`/server`)

```
server/
├── package.json
├── .env
├── src/
│   ├── app.js
│   ├── server.js
│   ├── config/
│   │   └── db.js
│   ├── models/
│   │   ├── userModel.js
│   │   ├── productModel.js
│   │   ├── userProductModel.js
│   │   └── orderModel.js
│   ├── services/
│   │   ├── authService.js
│   │   ├── productService.js
│   │   ├── cartService.js
│   │   └── orderService.js
│   ├── controllers/
│   │   ├── authController.js
│   │   ├── productController.js
│   │   ├── cartController.js
│   │   └── orderController.js
│   ├── routes/
│   │   ├── authRoutes.js
│   │   ├── productRoutes.js
│   │   ├── cartRoutes.js
│   │   └── orderRoutes.js
│   └── middleware/
│       └── authMiddleware.js
```

What each folder does:

Folder	Responsibility

`config/`	Database connection (connect once, share everywhere)
`models/`	Direct MongoDB queries — the ONLY layer that talks to the DB
`services/`	Business logic (hashing, validation, calculations)
`controllers/`	Reads HTTP requests, calls services, sends HTTP responses
`routes/`	Maps URL patterns to controller functions
`middleware/`	Runs between request and controller (e.g. auth check)

Client (`/client`)

```
client/
├── package.json
├── index.html
├── vite.config.js
├── src/
│   ├── main.jsx
│   ├── App.jsx
│   ├── api/
│   │   └── apiClient.js
│   ├── context/
│   │   ├── AuthContext.jsx
│   │   └── CartContext.jsx
│   ├── components/
│   │   ├── Navbar.jsx
│   │   ├── ProductCard.jsx
│   │   ├── CartItem.jsx
│   │   └── ProtectedRoute.jsx
│   ├── pages/
│   │   ├── HomePage.jsx
│   │   ├── ProductPage.jsx
│   │   ├── LoginPage.jsx
│   │   ├── RegisterPage.jsx
│   │   ├── CartPage.jsx
│   │   └── OrdersPage.jsx
│   └── styles/
│       └── index.css
```

Why this structure? The server follows **Clean Architecture** — each layer has ONE job. The data flows like this: **Route → Controller → Service → Model → Database**. This separation makes your code easier to read, test, and change.

API Endpoints Reference

Method	Endpoint	Auth?	Description
POST	`/api/auth/register`	No	Create a new user
POST	`/api/auth/login`	No	Login, returns JWT
GET	`/api/products`	No	Get all products
GET	`/api/products/:id`	No	Get single product
GET	`/api/cart`	Yes	Get user's cart items
POST	`/api/cart`	Yes	Add item to cart
PUT	`/api/cart/:id`	Yes	Update cart item quantity
DELETE	`/api/cart/:id`	Yes	Remove item from cart
POST	`/api/orders`	Yes	Place order (moves cart → order)
GET	`/api/orders`	Yes	Get user's order history

Let's Build It — Step by Step

Part 1 — Server Setup & Database Connection

Important: The entire server uses **ES Modules** (`import`/`export`) instead of CommonJS (`require`/`module.exports`). This is enabled by setting `"type": "module"` in `package.json`. When using ES modules in Node.js, you must include the `.js` extension in relative imports (e.g. `import { getDB } from '../config/db.js'`).

Step 1.1 — Initialize the Server Project

Create the `server` folder and initialize it:

```
mkdir server
cd server
npm init -y
npm install express mongodb dotenv cors bcrypt jsonwebtoken
npm install --save-dev nodemon
```

Add to `package.json`:

```
"type": "module",
"scripts": {
  "dev": "nodemon src/server.js"
}
```

Why? Every Node.js project starts with `npm init`. We add `"type": "module"` so we can use modern `import`/`export` syntax instead of the older `require()`/`module.exports`. The packages we install are the tools we need: `express` (web framework), `mongodb` (native driver — no Mongoose!), `dotenv` (loads secrets from `.env`), `cors` (lets React talk to our server), `bcrypt` (password hashing), `jsonwebtoken` (JWT auth).

Step 1.2 — Create the `.env` File

Create `.env` in the `server/` root:

```
PORT=5000
MONGODB_URI=mongodb://localhost:27017/ecommerce
JWT_SECRET=my_super_secret_key_change_this
```

Why? Sensitive values (DB URLs, secret keys) should **never** be hard-coded. The `.env` file keeps them separate. `dotenv` loads them into `process.env`.

Step 1.3 — MongoDB Connection (`config/db.js`)

Create `src/config/db.js`. This file connects to MongoDB **once** and shares that connection.

Your task: Write a module that:

1. Uses `import { MongoClient } from 'mongodb'`.

2. Creates a `MongoClient` instance using the URI from `process.env.MONGODB_URI`.
3. Exports a `connectDB` function that calls `client.connect()` and returns the database object.
4. Exports a `getDB` function that returns the already-connected database (so other files can use it).

Use named `export` for both functions.

Why? MongoDB connections are expensive to open. We connect **once** when the server starts, then reuse that connection for every request. `getDB()` gives all model files access to the same connection.

Step 1.4 — Express App Setup (`app.js`)

Create `src/app.js`. This configures Express but does **not** start the server.

Your task: Write a module that:

1. Imports `express` from `'express'` and `cors` from `'cors'`.
2. Creates an Express app.
3. Adds `cors()` middleware.
4. Adds `express.json()` middleware.
5. Uses `export default app` (don't call `app.listen` here).

Why? We separate Express *configuration* (`app.js`) from server *startup* (`server.js`). This is a best practice — it makes testing easier and keeps concerns separated.

Step 1.5 — Start the Server (`server.js`)

Create `src/server.js` — the entry point.

Your task:

1. Import `dotenv/config` at the top — this replaces the old `require('dotenv').config()` pattern. In ES modules, simply add `import 'dotenv/config';` as the first line and it will load your `.env` automatically.
2. Import `app` from `./app.js` and `connectDB` from `./config/db.js`.
3. Connect to the database, then start listening on the port from `.env`.

Checkpoint: Run `npm run dev`. You should see "MongoDB connected" and "Server running on port 5000". If you get a connection error, make sure your MongoDB server is running.

Part 2 — User Model & Authentication

This is the foundation everything else builds on. Without authentication, we can't have personalized carts or orders.

Step 2.1 — User Model (`models/userModel.js`)

Your task: Create a model file that exports functions to interact with the `users` collection.

Functions to implement:

- `findUserByEmail(email)` — finds a single user by email
- `createUser(userData)` — inserts a new user document
- `findUserById(id)` — finds a user by their `_id`

Remember: You're using the **native MongoDB driver**, so you'll use `getDB().collection('users')`. Use `import { ObjectId } from 'mongodb'` and `import { getDB } from '../config/db.js'`. Use named `export` for each function.

Why? The **Model layer** is the ONLY layer that talks to the database. If you ever switch databases, you only change this file — everything else stays the same.

Step 2.2 — Auth Service (`services/authService.js`)

Your task: Create a service that handles the *business logic* of authentication. Import `bcrypt` from `'bcrypt'`, `jwt` from `'jsonwebtoken'`, and the model functions from `'../models/userModel.js'`.

`registerUser(name, email, password)` should:

1. Check if a user with that email already exists (use the model).
2. If yes, throw an error.
3. Hash the password with `bcrypt`.
4. Create the user (use the model).
5. Return the created user (without the password).

`loginUser(email, password)` should:

1. Find the user by email.
2. If not found, throw an error.
3. Compare the password with the hash using `bcrypt`.
4. If wrong, throw an error.
5. Generate a JWT token containing `{ userId: user._id }`.
6. Return the token and user info.

Why? The **Service layer** contains business logic. It knows the *rules* (e.g. "passwords must be hashed") but doesn't know anything about HTTP requests or responses. It calls the Model for data and returns results to the Controller.

Step 2.3 — Auth Controller (`controllers/authController.js`)

Your task: Handle the HTTP request/response cycle.

- `register(req, res)` — calls the service, sends 201 on success or 400 on error.
- `login(req, res)` — calls the service, sends 200 with the token or 401 on error.

Why? The **Controller layer** is the glue between HTTP and business logic. It reads data from `req`, passes it to the service, and writes the response with `res`. It should be thin — no business logic here!

Step 2.4 — Auth Routes (`routes/authRoutes.js`)

Your task: Create a router that maps URLs to controller functions. Use `import express from 'express'`, create `const router = express.Router()`, and `export default router`.

Don't forget — go back to `app.js` and register the routes:

```
import authRoutes from './routes/authRoutes.js';
app.use('/api/auth', authRoutes);
```

Why? The **Routes layer** defines what URLs your server responds to. It maps `POST /api/auth/register` → `register controller function`. That's its only job.

Checkpoint: Test with Postman or curl:

```
# Register
curl -X POST http://localhost:5000/api/auth/register \
  -H "Content-Type: application/json" \
  -d '{"name":"Test User","email":"test@test.com","password":"123456"}'

# Login
curl -X POST http://localhost:5000/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{"email":"test@test.com","password":"123456"}'
```

You should get back a JWT token from the login endpoint. **Save this token** — you'll need it soon!

Step 2.5 — Auth Middleware (`middleware/authMiddleware.js`)

This is the gatekeeper. It protects routes that require a logged-in user.

Your task: Write middleware that:

1. Reads the `Authorization` header from the request.
2. Extracts the token (format: `Bearer <token>`).
3. Verifies it with `jwt.verify()` (import `jwt` from `'jsonwebtoken'`).
4. Attaches the decoded `userId` to `req.user`.
5. Calls `next()` if valid, sends 401 if invalid.

Use `export default` for the middleware function.

Why? Middleware runs *between* the request arriving and your controller handling it. When we add this to cart/order routes, only authenticated users can access them. Unauthenticated requests get a 401 error before they ever reach the controller.

Part 3 — Products

Step 3.1 — Product Model (`models/productModel.js`)

Your task: Export these functions:

- `getAllProducts()` — returns all products
- `getProductById(id)` — returns a single product by `_id`

Step 3.2 — Product Service (`services/productService.js`)

Your task: Create functions that call the model. For products the logic is simple — but we still go through the service layer to keep the architecture consistent.

Step 3.3 — Product Controller & Routes

Your task: Follow the same pattern as auth:

- `controllers/productController.js` — with `getAll` and `getOne` functions.
- `routes/productRoutes.js` — `GET /` and `GET /:id`.

Register the routes in `app.js`:

```
import productRoutes from './routes/productRoutes.js';
app.use('/api/products', productRoutes);
```

Checkpoint: Test `GET http://localhost:5000/api/products`. You should see your 12 products as JSON.

Note: Product data should already be in your database. If not, import the provided `products.json` file using MongoDB Compass (Add Data → Import JSON) or via the terminal:

```
mongoimport --db ecommerce --collection products --jsonArray --drop
--file products.json
```

Part 4 — Shopping Cart (Protected Routes)

This is where authentication becomes essential. Cart operations are **per user**, so every request needs a valid token.

Step 4.1 — UserProduct Model (`models/userProductModel.js`)

Your task: Export functions for the `userProducts` collection:

- `getCartItems(userId)` — gets all items for a user
- `findCartItem(userId, productId)` — checks if a product is already in the cart
- `addCartItem(item)` — inserts a cart item
- `updateCartItemQuantity(id, quantity)` — updates quantity by document `_id`
- `removeCartItem(id)` — deletes a cart item by document `_id`
- `clearCart(userId)` — removes all items for a user (used when placing an order)

Step 4.2 — Cart Service (`services/cartService.js`)

Your task: The cart service should:

- `getCart(userId)` — get cart items **with product details** (look up each product).
- `addToCart(userId, productId, quantity)` — if item exists, increase quantity; if not, add it.
- `updateQuantity(cartItemId, quantity)` — update item quantity.
- `removeFromCart(cartItemId)` — remove item.

Step 4.3 — Cart Controller & Routes

Your task: Create `controllers/cartController.js` and `routes/cartRoutes.js`.

Important: All cart routes must use `authMiddleware`. The userId comes from `req.user.userId` (set by the middleware).

Register in `app.js`:

```
import cartRoutes from './routes/cartRoutes.js';
app.use('/api/cart', cartRoutes);
```

Part 5 — Orders

Step 5.1 — Order Model, Service, Controller & Routes

Follow the exact same pattern: **model** → **service** → **controller** → **route**.

Order Model (`models/orderModel.js`) — functions to implement:

- `createOrder(orderData)` — inserts an order document
- `getOrdersByUser(userId)` — finds all orders for a user, sorted by date descending

Order Service (`services/orderService.js`) — `placeOrder(userId)` should:

1. Get the user's cart (use cart service).
2. If cart is empty, throw an error.
3. Calculate the total.
4. Create an order document with items, total, and `createdAt`.
5. Clear the user's cart.
6. Return the order.

Also implement `getUserOrders(userId)` — returns the user's orders.

Register in `app.js`:

```
import orderRoutes from './routes/orderRoutes.js';
app.use('/api/orders', orderRoutes);
```

Checkpoint — Full Server Test:

1. Register & login → get token
2. `GET /api/products` → see products
3. `POST /api/cart` with token + `{ "productId": "<id>", "quantity": 2 }` → item added
4. `GET /api/cart` with token → see cart with product details
5. `POST /api/orders` with token → order placed, cart emptied
6. `GET /api/orders` with token → see order history

Your server is complete!

Part 6 — React Client Setup

Step 6.1 — Initialize the React Project

```
cd ..
npm create vite@latest client -- --template react
cd client
npm install
npm install axios react-router-dom
```

Why Axios? Axios makes HTTP calls cleaner than `fetch`. It automatically parses JSON and lets us set default headers (like our JWT token) in one place.

Step 6.2 — API Client (`api/apiClient.js`)

Your task: Create an Axios instance with:

- `baseURL: 'http://localhost:5000/api'`
- A **request interceptor** that automatically adds the JWT token from `localStorage` to every request's `Authorization` header.

Why an interceptor? Without this, you'd have to manually add `Authorization: Bearer <token>` to *every single* API call. The interceptor does it automatically.

Step 6.3 — Auth Context (`context/AuthContext.jsx`)

Your task: Create a React Context that:

- Holds `user` and `token` state (initialized from `localStorage`).
- Provides a `login(email, password)` function that calls the API, stores the token, and updates state.
- Provides a `register(name, email, password)` function.
- Provides a `logout()` function that clears token and user.
- Provides an `isAuthenticated` boolean.

Why Context? Many components need to know if the user is logged in (Navbar, CartPage, ProtectedRoute). Instead of passing props through every level, Context makes auth state available **anywhere** in the component tree.

Step 6.4 — Cart Context (`context/CartContext.jsx`)

Your task: Create a React Context that manages all shopping cart logic in one place. This context should:

- Hold `cartItems` state (an array of cart items) and a `cartCount` derived value (total number of items).
- Provide a `fetchCart()` function that calls `GET /api/cart` and updates state. This should run automatically when the user is authenticated.
- Provide an `addToCart(productId, quantity)` function that:
    1. Checks if the user is authenticated (using AuthContext).
    2. If **not authenticated**, returns `{ redirectToLogin: true }` so the calling component can navigate to `/login`.
    3. If **authenticated**, calls `POST /api/cart` and then refreshes the cart.
- Provide an `updateQuantity(cartItemId, quantity)` function that calls `PUT /api/cart/:id` and refreshes the cart.
- Provide a `removeFromCart(cartItemId)` function that calls `DELETE /api/cart/:id` and refreshes the cart.
- Provide a `placeOrder()` function that calls `POST /api/orders`, clears the local cart state, and returns the result.
- Provide a `cartTotal` computed value (sum of price × quantity for all items).
- Clear the cart state when the user logs out. You can do this by watching the `isAuthenticated` value from AuthContext with a `useEffect`.

Export a `useCart()` hook for easy access.

Why a Cart Context? Without this, every component that touches the cart (ProductPage, CartPage, Navbar badge, etc.) would need its own API calls and its own copy of the cart state. This leads to duplicated logic and out-of-sync data. By centralizing cart logic in a context, you get: a single source of truth for cart data, one place to change if the API changes, and automatic updates across all components when the cart changes. This is the same pattern used by real e-commerce apps.

How does it connect to AuthContext? CartContext *consumes* AuthContext — it checks `isAuthenticated` before making API calls, and clears cart data on logout. This means CartContext must be rendered *inside* AuthProvider in your component tree.

Step 6.5 — Router & App Structure (`App.jsx`)

Your task: Set up React Router with these routes:

Path	Component	Protected?
`/`	HomePage	No
`/products/:id`	ProductPage	No
`/login`	LoginPage	No
`/register`	RegisterPage	No
`/cart`	CartPage	**Yes**
`/orders`	OrdersPage	**Yes**

Wrap the entire app in `BrowserRouter` → `AuthProvider` → `CartProvider` (in that order). The order matters because CartContext depends on AuthContext, so `CartProvider` must be nested inside `AuthProvider`.

Also update `main.jsx`:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import './styles/index.css';

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Step 6.6 — ProtectedRoute Component (`components/ProtectedRoute.jsx`)

Your task: Create a component that checks `isAuthenticated` from AuthContext. If the user is not authenticated, redirect to `/login`. Otherwise, render the children.

Part 7 — React Pages

Step 7.1 — Navbar (`components/Navbar.jsx`)

Your task: Create a navigation bar that:

- Always shows a link to Home (`/`).
- If **not** logged in: shows Login and Register links.
- If **logged in**: shows Cart link **with item count badge** (from CartContext's `cartCount`), Orders link, user name, and Logout button.

Step 7.2 — Login & Register Pages

LoginPage (`pages/LoginPage.jsx`): A form with email and password. On submit, call `login()` from AuthContext. On success, navigate to `/`. On error, show the error message.

RegisterPage (`pages/RegisterPage.jsx`): A form with name, email, and password. On submit, call `register()`. On success, navigate to `/login`. On error, show the error message.

Now build RegisterPage yourself following the same pattern!

Step 7.3 — HomePage & ProductCard

HomePage (`pages/HomePage.jsx`):

1. Fetches all products on mount using `useEffect` and `apiClient.get('/products')`.

2. Renders a grid of `ProductCard` components.

ProductCard (`components/ProductCard.jsx`): Shows the product image, name, price, and links to `/products/:id`.

Step 7.4 — ProductPage (Single Product + Add to Cart)

This is a **key** feature: the "Add to Cart" button must redirect to login if the user is not authenticated.

Your task: Create a page that:

1. Gets the product ID from the URL using `useParams()`.
2. Fetches the product on mount.
3. Shows product details (image, name, description, price).
4. Has an "Add to Cart" button that:
    - Calls `addToCart(productId, 1)` from **CartContext**.
    - If the result contains `{ redirectToLogin: true }` → navigates to `/login`.
    - Otherwise → shows a "Added to cart!" success message.

Why this pattern? The page doesn't need to know about authentication or API calls — it just asks CartContext to add an item, and CartContext handles everything. If the user isn't logged in, CartContext tells the page to redirect. This keeps the page simple and the logic centralized.

Step 7.5 — CartPage

Your task: Create a page that uses **CartContext** to display and manage the cart:

1. Gets `cartItems`, `cartTotal`, `updateQuantity`, `removeFromCart`, and `placeOrder` from CartContext.
2. If the cart is empty, shows an "Your cart is empty" message.
3. For each item, shows: product name, unit price, quantity with +/- buttons, and line subtotal.
4. The +/- buttons call `updateQuantity(cartItemId, newQuantity)` from CartContext.
5. A "Remove" button calls `removeFromCart(cartItemId)` from CartContext.
6. Shows the `cartTotal` at the bottom.
7. A "Place Order" button calls `placeOrder()` from CartContext, then navigates to `/orders`.

Why use CartContext here? Notice how the page has zero API calls — it only talks to CartContext. If you later change the API structure, you only update CartContext and every page stays the same.

Step 7.6 — OrdersPage

Your task: Create a page that fetches and displays the user's order history. Each order should show the date, list of items, and total.

Part 8 — Final Steps

Step 8.1 — Global Styles

Create `src/styles/index.css`:

```
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body {
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto,
sans-serif;
  background-color: #f5f5f5;
  color: #333;
}

a {
  color: #2c3e50;
}
```

Step 8.2 — Run Both Projects

Terminal 1 — Server:

```
cd server
npm run dev
```

Terminal 2 — Client:

```
cd client
npm run dev
```

Final Checklist

Test each feature end-to-end:

#	Test	Expected Result
1	Visit home page	See 12 product cards
2	Click a product	See product details
3	Click "Add to Cart" (not logged in)	Redirected to login page
4	Register a new account	Success, redirected to login
5	Login	Success, redirected to home, Navbar shows your name
6	Click "Add to Cart" (logged in)	"Added to cart!" message, Navbar cart count updates
7	Go to Cart	See item with quantity controls
8	Change quantity, remove item	Cart updates correctly
9	Place Order	Redirected to orders page, cart is now empty
10	Click Logout	Navbar shows Login/Register again

Architecture Summary

```
REACT CLIENT
=============
AuthContext      -->  stores token + user  -->  apiClient sends token with every request
CartContext      -->  stores cart state    -->  provides addToCart, removeFromCart, placeOrder
Pages            -->  use contexts         -->  render data, no direct API calls for cart
ProtectedRoute   -->  checks isAuthenticated  -->  redirects guests
```

```
Provider nesting:  BrowserRouter > AuthProvider > CartProvider > App

                              |
                              |  HTTP Requests (with JWT in header)
                              v

EXPRESS SERVER
==============
Routes           -->  define URL patterns
Middleware       -->  authMiddleware verifies JWT, attaches userId to
req
Controllers      -->  read req, call service, write res
Services         -->  business logic (hash passwords, validate,
calculate totals)
Models           -->  direct MongoDB queries (find, insert, update,
delete)
MongoDB          -->  users, products, userProducts, orders collections
```

Bonus Challenges (Optional)

If you finish early, try adding:

1. **Search & Filter** — Add a search bar on the home page that filters products by name or category.
2. **Stock Validation** — Check product stock before adding to cart. Show "Out of Stock" when stock is 0.
3. **Pagination** — Only show 6 products per page with Next/Previous buttons.
4. **Loading States** — Show spinners while data is being fetched.
5. **Error Boundaries** — Handle network errors gracefully with user-friendly messages.

Good luck and have fun building!