Project: Full-Stack E-Commerce Application

Overview

In this project you will design and build a **complete e-commerce web application** consisting of:

1. A **backend REST API** click me built with **FastAPI** and persisted in a relational database using **SQLite** or **Supabase**.
2. A **frontend single-page application (SPA)** built with **React + TypeScript**.

The application will allow users to browse a catalog of products, manage a shopping cart, place orders, and view their order history. You will gain hands-on experience integrating a modern frontend with a Python-based backend, handling authentication, and deploying a full-stack application.

This project is worth **40% of the final grade**. To pass you must obtain a **minimum score of 4.0/10.0**.

Goal

Build and deploy a small but functional **e-commerce platform** with the following capabilities:

Area	What you will implement
Product catalog	Public browsing, search/filter, detailed product view
Shopping cart	Client-side cart with quantity management and server-side validation
User accounts	Registration, login, and JWT-based authentication
Order management	Place orders (with stock deduction), view order history
Deployment	Publicly accessible backend and frontend URLs

Data models

`User`

Field	Type	Notes
`id`	int	PK
`email`	str	unique, indexed
`password_hash`	str	stored hash
`is_admin`	bool	default False

Field	Type	Notes
`created_at`	datetime	default now

`Product`

Field	Type	Notes
`id`	int	PK
`title`	str	required
`slug`	str	unique, URL-friendly
`description`	str	text
`price_cents`	int	integer to avoid float issues
`currency`	str	default `"USD"`
`stock`	int	current inventory
`created_at`	datetime	default now
`updated_at`	datetime	auto-update

`Order`

Field	Type	Notes
`id`	int	PK
`user_id`	int	FK -> User
`status`	str	`"pending"`, `"paid"`, `"cancelled"`
`total_cents`	int	total order amount
`currency`	str	default `"USD"`
`created_at`	datetime	default now

`OrderItem`

Field	Type	Notes
`id`	int	PK
`order_id`	int	FK -> Order
`product_id`	int	FK -> Product
`unit_price_cents`	int	snapshot of product price
`quantity`	int	must be >= 1

You may modify this or add more tables if you consider it necessary. Your decisions should be explained in the README.md

References

sqlmodel

API structure

Route group	Prefix	Purpose
Health	`/health`	basic check
Auth	`/auth`	register, login, user info
Products	`/products`	browse, view product
Checkout	`/checkout`	validate cart before order
Orders	`/orders`	create and view orders

You may modify this. Your decisions should be explained in the README.md

Endpoints specification

Health check

`GET /health`

- Returns `{status: "ok"}`.

Frontend

You must build a **single-page application (SPA)** using **React + TypeScript**.

The frontend is responsible for:

- Public browsing of products.
- Managing a client-side shopping cart.
- Interacting with the backend API for checkout and orders.
- Handling user registration and login flows so that only **logged-in users** can complete the checkout.

Requirements

1. **Tech stack**
   - Use **React** with **TypeScript**.
   - You may use **React Router**.
2. **Minimum screens / views**

   You must implement at least the following views with the corresponding backend functionality. You are free to choose how they look and how you organize components.

   - **Product catalog**
     - Show a list/grid of products obtained from the backend endpoint.
     - Include some form of **filtering or searching** (e.g. by title or text query `q`).
   - **Product detail**

    - Shows the information of a single product.
    - Allow the user to **add the product to the cart** with a configurable quantity.
  - **Login / Register views**

3. **Other screens**

   The following additional screens are required:

   - **Cart**
     - Shows the current items that the user has added.
     - Allows changing quantities and removing items.
     - Shows the subtotal per item and the cart total.
     - Includes a button to **validate / preview checkout**.
   - **Checkout / Order confirmation**
     - Show the final, validated total and any invalid items.
     - Allows the user to **confirm the order only if they are logged in**.
   - **Orders history view**

4. **API integration**

   - All product, order data and so on, must come from your **own backend API**.
   - Handle basic error states: show a message when a request fails (network error, 4xx/5xx).

5. **UX and responsiveness**

   - The layout should be usable on both **desktop and mobile**.
   - Navigation between main views must be clear (e.g. a header with links to "Home", "Cart", and optionally "My orders" / "Login").
   - Loading and empty states should be handled (e.g. show a spinner or message while data is loading, display "No products found" when appropriate).

Project evaluation criteria

This project will be evaluated based on the following aspects:

- **Correct use of input and output models**: Proper implementation of Pydantic schemas for request validation and response formatting. Each endpoint must define clear input/output contracts with appropriate type hints and validation rules.
- **Code organization and clarity**: Well-structured codebase with logical separation of concerns (models, routers, services, components, pages, ...). Backend and frontend code should be readable, maintainable.
- **API design quality**: Well-designed endpoints with appropriate HTTP methods, meaningful route names, proper use of query parameters (pag-

ination, filtering, sorting), correct status codes, and consistent response formats.

- **Data validation and security**: Proper validation of all inputs, secure password hashing, JWT implementation, and protection against common security vulnerabilities.
- **Database operations**: Correct use of SQLModel for data persistence, including proper relationships, transactions, and error handling.
- **Business logic correctness**: Accurate implementation of cart validation, order creation, stock management, and user authentication flows.
- **Frontend UX and API integration**: Clear navigation between pages, correct consumption of the backend API, proper handling of loading/error states, and a responsive layout.
- **Optional admin dashboard (bonus)**: If you implement an admin dashboard to manage users and products (see "Bonus: Admin dashboard" below), the quality and completeness of that feature can contribute up to **+3 extra points**, without exceeding the maximum grade of **10.0**.

Deployment options

- **Vercel**
- **Render**

Deliverables

You must submit a **Git repository** containing both the backend and the frontend for your e-commerce application.

1. **Public deployments (live URLs)**
   - A publicly accessible URL for the **backend API**.
   - A publicly accessible URL for the **frontend SPA**.
   - Both URLs must be clearly indicated in the README.
2. **Source repository**
   - Public Git repository containing **all source files** for backend and frontend.
   - Organize the code in a clear, logical structure (for example, a `backend/` and `frontend/` folder or a similar convention that is easy to understand).
   - Include any configuration files required to run the project locally (e.g. `requirements.txt` / `pyproject.toml`, `package.json`, Vite config, etc.).
3. **README.md** (at the root of the repository)

The README must include at least:

- Short project overview (what your e-commerce store does, any special features).
- Backend and frontend **live URLs**.
- Instructions to **run the backend and frontend locally** (dependencies and commands).
- Description of any **design/architecture decisions** you made (e.g. how you modelled the database, how you structured React state, any notable trade-offs).
- Brief explanation of any **extensions** beyond the minimum requirements (e.g. product images, categories, admin features, improved search, etc.).

Grading

This project contributes **40%** to the final course grade. The evaluation will focus on the following aspects:

- **Backend implementation and API design**
  - Correct use of FastAPI, SQLModel, and the specified data models.
  - Clear, well-designed endpoints and input/output models.
  - Correct business logic for cart validation, order creation, and stock management.
- **Frontend implementation and user experience**
  - Correct use of React, TypeScript, and Vite.
  - Proper integration with the backend API and cart/order flows.
  - Usable, responsive UI with clear navigation and sensible UX.
- **Code quality and organization**
  - Clean, readable code with meaningful names and comments where needed.
  - Logical structure of modules, components, and folders on both backend and frontend.
- **Deployment and reliability**
  - Working, publicly accessible deployments for both backend and frontend.
  - Basic error handling and resilience (e.g. graceful handling of failed requests).
- **Documentation and reflection**
  - Clear README with setup instructions, URLs, and design decisions.
  - Evidence that you understand and can explain your own implementation choices.

To **pass** this project component you must achieve at least **4.0/10.0 overall** on this assignment.

Bonus: Admin dashboard (optional)

You can earn up to **+3 extra points** (without exceeding a final grade of **10.0**) by implementing an **admin dashboard** to manage the content of your application. This part is **optional** and should only be attempted after the core requirements are complete.

Ideas for what the admin dashboard may include (you do not need to implement all of them):

- Restricted access for admin users only (e.g. `is_admin = true`).
- A simple **web interface** to:
  - List, create, update, and deactivate products.
  - Manage users (e.g. view users, promote/demote admin status).
- Backend endpoints to support these operations (e.g. `/admin/products`, `/admin/users`, or similar routes of your choice).
- Basic safeguards (e.g. avoid deleting data that would break existing orders; prefer soft-deletes or status flags).

You are free to decide the exact routes, UI layout, and data model details, as long as:

- The admin dashboard is clearly separated from the public storefront.
- Only authenticated admin users can access admin functionality.
- The behavior and design choices are documented in the README.

Submission instructions

Submit on the PDU: **GitHub repository**

The name of the zip should be id_number_ecommerce.zip, anything else will not be accepted.

Academic integrity

- This is an **individual** assignment. You may discuss ideas with peers, but all code and content you submit must be your own.
- AI-assisted tools may be used for **non code generative** tasks (e.g., concepts explanation) as long as you understand and can explain every line of submitted code.

Timeline

- **Due date:** 18 January. Refer to the course PDU for the official deadline.

Acceptance criteria (must all be met)

1. **Backend and frontend are both deployed** and reachable at public URLs.

2. The backend exposes the required API endpoints and persists data in a real database (e.g. SQLite).
3. The frontend is implemented as a React + TypeScript SPA and consumes your own backend API.
4. Users can perform the mentioned in **Minimum screens / views** flows end-to-end
5. The repository is clean and organized, with a complete README that documents URLs, setup steps, and key design decisions.