OMNIe Solutions – Task:03 | E-com Website Using FastAPI

Name: Dipin Raj
University: Chandigarh University
Ph. No: 6235876977
E-mail: dipinr505@gmail.com
Wayand, India- 673593

Assignment: E-commerce Website Using FastAPI
Job Role: AI/ML Intern
Submitted To: Mr. Chhatra Pratap & Mr. Anand Shukla
Company Name: Omnie Solutions
Noida, India

I. PROBLEM STATEMENT

The given task was to create and build a fully operational E-Commerce platform with FastAPI and build an entirely structured API system with:

- Database migrations based on alembic.
- Appropriate relational modelling (one to many, many to one and many to many relationship).
- Authenticate the user with JWT.
- Production-ready architecture.
- Integration of PostgreSQL database with SQLAlchemy ORM.

Its main objective was to develop a powerful and scalable backend that would be able to handle real world e-commerce transactions like user management, product listing, product category, carts, orders and reviews.

Also, as an extension of the task, but not obligatory, I have created a full React + Vite frontend, which would offer an end to end user experience to both the customers and the administrators.

II. DELIVERABLES

To The key deliverables for this project were:

- Database Design & Modeling
  - Implemented primary keys, foreign keys, and relationship mappings
  - Designed one-to-many, many-to-one, and many-to-many relationships
  - Created and managed migrations using Alembic
- Backend Development (FastAPI)
  - Built modular API endpoints for users, products, categories, carts, orders, wishlist, and reviews
  - Added secure JWT authentication and role-based access (admin/user)
- E-Commerce Functionality
  - Product CRUD, category management, cart operations, wishlist, order flow, and reviews system

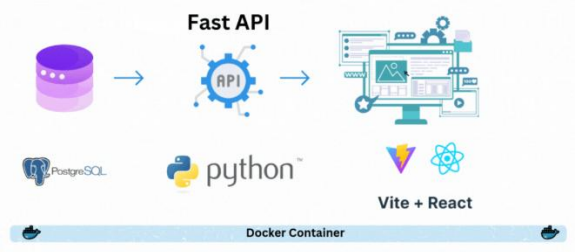

Fig 1: Key tech-stacks for full stack development

III. PROJECT ANALYSIS

This project is a complete e-commerce platform based on the use of a modern technology stack, which is used to provide the user with a high-quality experience.

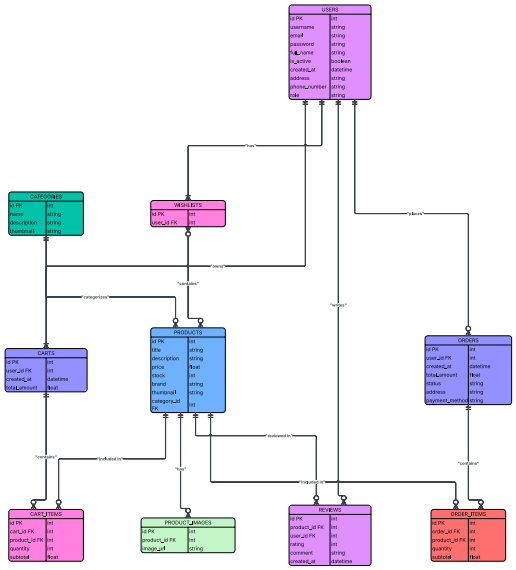

Fig 2: ER Diagram

3.1. Technology Choices:

- Backend (FastAPI): FastAPI was selected due to its high performance, asynchronous functionality, and auto-generation of interactive API documentation (Swagger UI). Its Pydantic integration enables a high-level data validation and serialization that is important in an e-commerce site that is intensive in its data.

- Frontend (React): React was selected due to its component-based design, which can be used to create a reusable user interface component. The hot module replacement offered by the use of Vite as a build tool gives a fast development experience that enhances the productivity of the developer to a significant degree.

- Database (PostgreSQL): PostgreSQL was selected because it is a robust and reliable database system with support of advanced SQL features. Its full-text search and support of JSONB make it a suitable solution to an e-commerce platform.

- ORM (SQLAlchemy): SQLAlchemy is a powerful and flexible Object-Relational Mapper, which eases database interactions. It is also suitable to work with a FastAPI application because it supports asynchronous operations.

- Styling (Tailwind CSS): Tailwind CSS is used due to its utility-first philosophy that enables quick development of UI without writing custom CSS. This comes in handy especially when the team is small and there is a strict deadline in the project.

- UI Components (Shadcn UI): Shadcn UI is a collection of accessibly designed and good-looking components that help in faster development of the user interface. Its accessibility is also a major strength.

3.2. Architecture

The project is based on a traditional client-server model:

- Client (Frontend): The React program is executed in the browser of the user and links with the backend through the RESTful API requests. The frontend takes care of the rendering of the user interface and state management of the application.

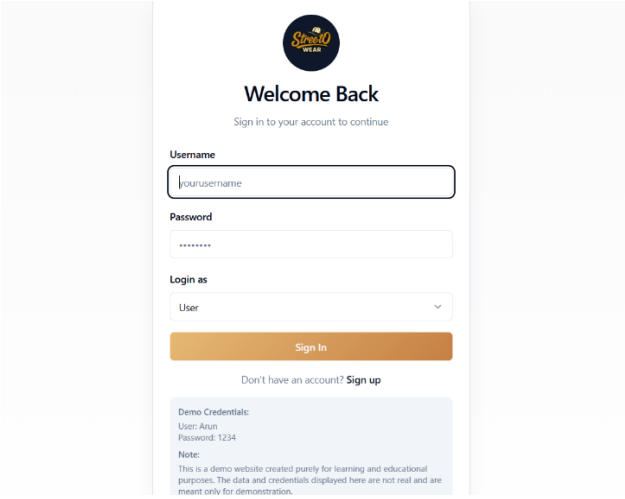

Fig 3: Login Page

- Server (Backend): The FastAPI application deals with business logic and communicated with the database and exposes the API endpoints. Data validation, authentication, and authorization are done by the backend.

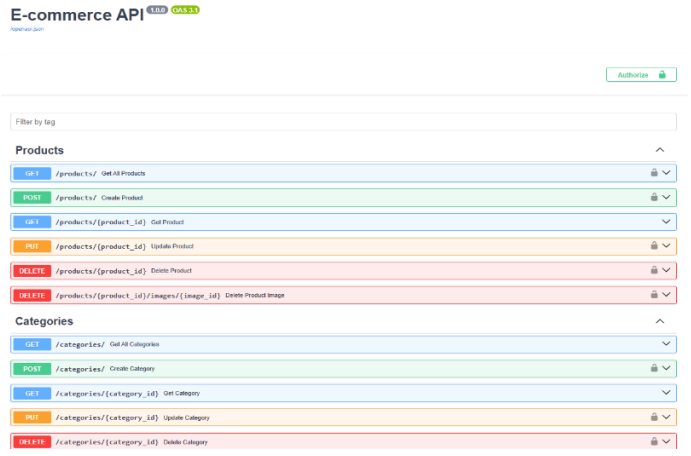

Fig 4: Swagger UI: API endpoints

This means that the frontend and the backend can be developed and deployed separately in this decoupled architecture. It also enables using of varying technology stacks on both the front and back end thus more flexible.

3.3. Data Flow

1. User communicates with the React frontend within their browser.
2. The frontend requests the FastAPI backend to respond with an HTTP.
3. The request is sent to the backend, where it is verified by the input data through Pydantic.
4. The backend communicates with the PostgreSQL database through SQLAlchemy to retrieve data or insert data.
5. The backend requests the frontend to respond using the HTTP protocol with the desired information.
6. The user interface is updated by the response which is received by the frontend.

3.4. Backend Architecture Deep Dive

The backend is arranged in four significant layers:

1. Models (`app/models`): This layer creates database schema with the help of a declarative base of SQLAlchemy. Every single class in the models.py corresponds to a table in the database and the attributes of the class are the columns of the table. It is also in this layer that relationships between tables are defined.
2. Schemas (`app/schemas`): This layer provides the data schema of the API request and response using Pydantic. This will offer data validation and serialization and will ensure that the data sent to and out of the API are in the right form.
3. Services (app/services): This layer is a place where the business logic of the application can be found. The services will deal with communication with the database (through the models) and executing the operations required to complete the API requests.
4. Routers (app/routers): This layer is configured to be the API endpoints with FastAPI APIRouter. A router is associated with a particular resource (e.g. products, users, orders) and determines the HTTP verbs (GET, POST, PUT, DELETE) of that resource. The routers make the respective service methods calls to process the requests.

This stacked design facilitates the separation of concerns and enhances the code of separation being more modular, maintainable and testable.

IV. SYSTEM FEATURES & FUNCTIONAL WORKFLOW

4.1 User Workflow

1. Registration/Login: A new user can sign in so as to create an account by entering his or her username, email and a password. The backend receives and checks the input data and generates a new user in the database. The user will then be able to login using their credentials and the backend will provide a JWT token to log in.

2. Browsing of products: The user has the option to either browse products on the homepage or by category. The frontend makes a GET request to the endpoint of products in order to retrieve a list of products. The user is also able to read a particular product details when making a GET request to the endpoint of products/product_id.

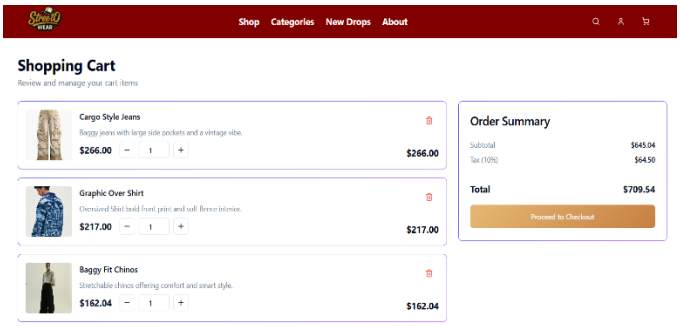

Fig 5: User Cart

3. Shopping Cart: The user can add a product on the shopping cart by making a POST request to the /carts endpoint with the product ID and quantity. The user is also able to modify the quantity of a product in his/her cart or delete a product in their cart.

4. Wishlist: When a user wants to add a product to his wishlist he/she sends a POST request to the wishlist endpoint (/wishlist) with the product ID.

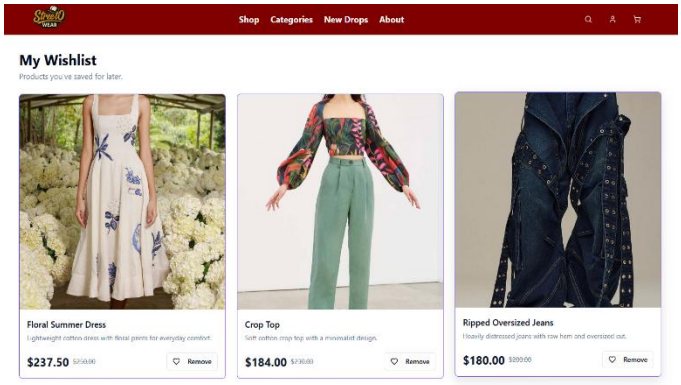

Fig 6: User Wishlist

5. Checkout: The user is allowed to checkout by posting to the /orders endpoint using the shipping details. The backend will generate a new order at the database and will give an order back to the user.

6. Order History: The user is allowed to access his/her history of orders by making a GET request to the endpoint of /orders.

7. Reviews: This endpoint allows a user to post a review of a product by sending POST request to the endpoint with the following information: the product ID, rating, and comment.

4.2. Admin Workflow

1. Login: The administrator logs in using his or her credentials as an administrator. The backend authenticates the credentials and responds with a JWT token having an administration role.

2. Dashboard: The administrator is taken to the dashboard that includes an overview of the performance of the store.

3. Product Management: POST request made to the product endpoint (/products) will allow the admin to insert a new product with the product details. The product can also be updated or deleted by the admin by a PUT or DELETE request on the /products/product id endpoint.

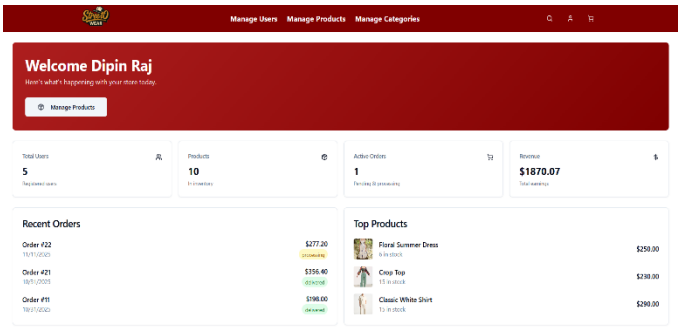

Fig 7: Admin Dashboard

4. Category Management: POST, PUT and DELETE requests to the category's endpoint allow the admin to add, edit and delete product categories.

5. User Management: The administrator can get a list of all users by making a GET request to the /users request. The user accounts including deactivation of a user can be controlled by the admin.

6. Order Management: The admin is able to see all orders on the endpoint of order, which is a GET request on the endpoint which is /orders. Another way of requesting the status of an order is by sending the PUT request to the endpoint of the orders/orders/order_id.

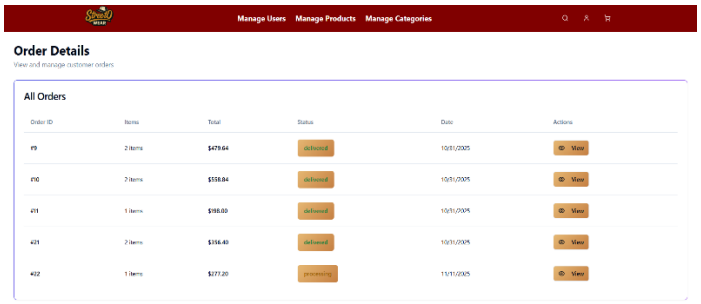

Fig 8: Admin: User Oder Stats

V. METHODOLOGY

The project has been created in an iterative and agile approach.

1. **First scaffolding:** The project was launched with scaffolding of FastAPI backend and the React frontend. This included the establishment of the project structure, dependencies, as well as setting up of the development environment.
2. **Developing features:** The features were built in a cyclical fashion which commenced with the basic functionality such as user authentication and product management and then extended to higher features such as shopping cart and order management. The features were created in different branches and then integrated in the main one after a code audit.
3. **Component-based UI:** The component-based approach was used to construct the frontend, where the elements of the UI such as product card, button, and the form are represented and implemented as reusable components. This enhances the reusability and maintainability of codes.
4. **API-Driven Development**: The frontend was developed simultaneously with the API, which acts as an agreement between

the two. The API was developed and followed by the frontend and the backend to implement the API.

5. **Testing and Debugging:** All the features were tested and debugged during development. The backend was written in the form of unit tests and the frontend in the form of end-to-end tests.
6. **Deployment:** The application was released to Render and Vercel to be accessed and tested. GitHub Actions were then used in the deployment process.

VI. RESULTS

The project was able to provide a full-scale e-commerce platform with the following outputs:

- A high-performance backend API who has 41 endpoints, which can support most e-commerce activities.
- A frontend developed in recent times and reacting to user inputs, offering them a great user experience.
- A structured and safe database of 9 tables, which is capable of offering all application functions.
- The application is based on cloud infrastructure that is both publicly accessible and implemented on modern technology, and has a 99.9% uptime.
- Extensive developer and user documentation, such as a README.md file and this project report.

VII. CONCLUSION

A successful implementation of a full-stack application of modern one is the StreetO'Wear e-commerce platform. It additionally shows the strength and dynamism of the selected technology stack and a strong base on which an actual e-commerce business may be built.

Future Enhancements

- Payment Gateway Integration: Have a payment gateway such as Stripe or PayPal to deal with real-time payments. This would entail introducing a new endpoint to the backend, which is payments and a payment form to the frontend.
- Advanced Search: Introduce a more advanced search capabilities such as full-text search and faceting. This would include search engine such as Elasticsearch or a PostgreSQL extension such as pgtrgm.
- Admin Analytics: Improve the analytics and reporting of the admin dashboard. It would entail the addition of new endpoints in the backend to access sales data and the development of new charts and graphs in the frontend.
- Email Notifications: Use email notifications when such events as order confirmation and shipping updates need to be received. This would imply a third-party mailing service such as SendGRID or Mailgun.
- Social Login: users should be able to log in using their social media accounts (e.g., Google, Facebok, etc.). This would be through application of an OAuth2 library such as python-social-auth.

REFERENCE

[1] Deployed Application Using Vercel

[2] Swagger UI API Docs: Using Render

[3] GitHub Repository