BHAO.PK CENTRALIZED E-COMMERCE PRODUCT SEARCH AND PRICE MONITORING ENGINE

MUHAMMAD AHAD ALI KHAN

SHERYAR KARIM

MUHAMMAD HARIS JAMALI

Fall-2025

Department of Computer Science

Capital University of Science & Technology, Islamabad

Submission Form for Final-Year

PROJECT REPORT

Version	V 1.0

NUMBER OF MEMBERS	03

TITLE	Bhao.pk Centralized E-commerce Product Search and Price Monitoring Engine

SUPERVISOR NAME	DR. Najam Aziz

MEMBER NAME	REG. NO.	EMAIL ADDRESS
Muhammad Ahad Ali Khan	BCS213129	alik123kh@gmail.com
Sheryar Karim	BCS221074	sheryarkarim65@gmailgmail.com
Muhammad Haris Jamali	BCS221027	harisshaheen09@gmail.com

Supervisor's Signature

APPROVAL CERTIFICATE

This project, titled as “Bhao.pk Centralized E-commerce Product Search and Price Monitoring Engine” has been approved for the award of

Bachelors of Science in Computer Science

Committee Signatures:

Supervisor: _____

(Dr. Najam Aziz)

Project Coordinator: _____

(Mr. Ibtisam Zia)

Head of Department: _____

(Dr. Masroor Ahmed)

DECLARATION

I/We, hereby, declare that “No portion of the work referred to, in this project has been submitted in support of an application for another degree or qualification of this or any other university/institute or other institution of learning”. It is further declared that this undergraduate project, neither as a whole nor as a part thereof has been copied out from any sources, wherever references have been provided.

MEMBERS’ SIGNATURES

ACKNOWLEDGEMENTS

Executive Summary

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter introduces the Bhao.pk project, describing its problem statement, objectives, and the technologies used to implement it. It also presents the scope, significance, and development plan of the proposed system. By the end of this chapter, the reader will understand how the project aims to simplify online shopping decisions in Pakistan through intelligent price comparison and alert mechanisms.

1.1. Project Introduction

Bhao.pk is an intelligent web and mobile platform that aggregates prices of products listed across multiple Pakistani e-commerce sites. It enables users to quickly identify the lowest price for a specific product and receive instant alerts when prices drop below a set threshold. The system collects product and pricing data through automated web scrapers, cleans and normalizes the data, and then displays results on an interactive dashboard.

The project's core purpose is to give users transparent access to price information that is otherwise scattered across different retailers. A single search query such as *"70W USB-C Charger"* will return aggregated listings from major online stores like **Daraz, Shophive, Mega.pk, PriceOye**, and **BTech**. The backend clusters identical or similar products using fuzzy text matching to eliminate duplicates and inconsistencies.

The platform will also maintain a historical record of price changes, allowing users to visualize price trends over time and set up alerts when prices fall below their target value. The entire system is designed to be cross-platform: the same codebase will generate both a responsive web application and a mobile app.

.

1.2. Existing Examples / Solutions

Existing global price comparison systems show that price aggregation is highly valuable for consumers, but none are specialized for Pakistan's fragmented e-commerce ecosystem.

Google Shopping aggregates product listings from thousands of retailers worldwide, offering side-by-side comparisons and filtering by category or brand. However, it depends on merchant data feeds and lacks real-time integration with local Pakistani stores. Its algorithms are optimized for structured APIs, which limits adaptability to dynamic regional websites.

Idealo, one of Germany's leading price comparison engines, provides detailed price histories, vendor ratings, and automated alerts when prices change. It uses a mix of API and web-scraping methods but operates exclusively in European markets. While technically advanced, it cannot process PKR pricing or local vendor structures.

Local Attempts in Pakistan include smaller startups and online deal-sharing communities that post offers manually on social media platforms like Facebook or WhatsApp. These efforts lack automation, consistent data collection, and historical tracking. As a result, users must still visit multiple sites individually to verify prices and authenticity.

***i*: Table 1.1: Comparison with Existing Examples**

System/ Platform	Core Functionality	Data Source	Limitations	Unique Value of Bhao.pk
Google Shopping	Aggregates global product listings, enabling real-time price and availability comparison across thousands of retailers.	Merchant-supplied APIs and global data feeds.	No integration with Pakistani vendors; does not support PKR-based pricing.	Serves as a global benchmark for automated product aggregation and intelligent ranking.
Idealo (Germany)	Provides price tracking, vendor ratings, and discount alerts across European retailers.	Retailer APIs and automated web crawlers across EU marketplaces.	Limited to European markets; cannot process PKR or local Pakistani vendors.	Demonstrates how historical price tracking builds trust and empowers consumers.
Local Attempts (Pakistan)	Manual deal sharing through Facebook, WhatsApp, and Telegram communities.	User-posted, unstructured community data.	No automation, centralized dashboard, or consistent price tracking.	Highlights the need for a unified, automated, Pakistan-specific price comparison ecosystem like Bhao.pk.

1.3 Business Scope

Bhao.pk has strong market potential in Pakistan, where online shopping continues to grow but lacks transparent price comparison tools:

1.3.1 The Intended Audience

- **Everyday Online Shoppers:** Looking for lowest prices without visiting multiple websites.
- **Students and Budget Buyers:** Seeking affordable tech, accessories, or daily items.
- **E-commerce Analysts:** Can study price trends and market volatility.
- **Retailers:** Can analyze competitor pricing dynamically.

1.3.2 Use in Commercial Settings

- Integration into affiliate marketing networks.
- Tools for businesses to track competitor prices and adjust strategies.
- Integration with fintech cashback or coupon systems.
- Academic use in data analytics and information retrieval projects.

1.3.3 Models of Revenue

- **Affiliate Commissions:** Redirect users to retailers through tracked links.
- **Ad-Based Revenue:** Sponsored listings or priority placement for vendors.
- **Subscription Plans:** Users pay for extended alerts and historical data exports.

- **API Access:** Businesses pay to access aggregated pricing data.

1.4. Useful Tools and Technologies

The development of **Bhao.pk** relies on a combination of web technologies and data-processing frameworks to ensure smooth cross-platform performance and efficient information retrieval.

The system is built using **React with React Native (Expo)**, which allows both the web dashboard and the mobile application to share the same codebase. This framework ensures responsive interfaces, real-time updates, and reduced development effort compared to maintaining separate apps.

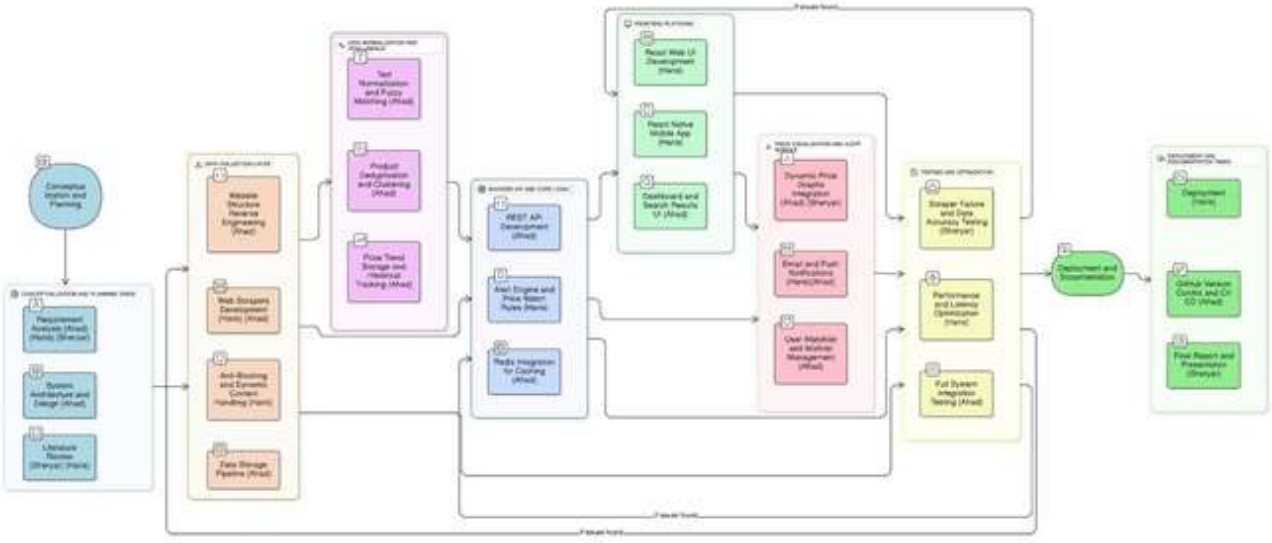
For backend services, **Node.js with Express** provides a lightweight, scalable environment capable of handling multiple concurrent API requests efficiently. It manages data flow between the scrapers, database, and user interface, forming the communication core of the system.

Automated data collection is powered by **Scrapy and BeautifulSoup**, which extract structured product and pricing data from e-commerce websites. These libraries handle HTML parsing, data cleaning, and link traversal, enabling continuous updates without manual input.

The project stores and manages information using **PostgreSQL**, an open-source relational database that supports large-scale data queries, indexing, and time-series analysis. It ensures reliability and fast retrieval for price tracking, user alerts, and historical analytics.

1.5. Project Work Break Down

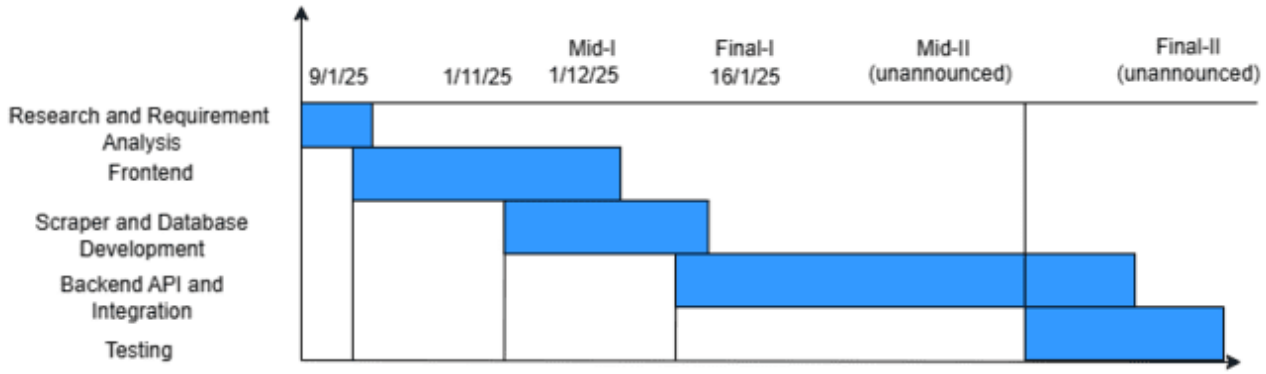
The project breakdown structure is shown in Figure 4



i: Figure 1.1: Project Work Break down

1.6. Project Time Line

The proposed project timeline is shown in the following the figure.



ii: Figure 1.2: Project Timeline

Chapter 2

Requirement Specification and Analysis

Requirements analysis is a process of determining user expectations for a new or modified product. These features, called requirements, must be quantifiable, relevant and detailed. In software engineering, such requirements are often called functional specifications. In Chapter 2 we will enlist the functional and non-functional requirements and model functional requirements in the form of use case model.

2.1. Functional Requirements

A functional requirement defines a function of a system or its component. Functional requirements may be calculations, technical details, data manipulation and processing and other specific functionality that define what a system is supposed to accomplish.

ii: Table 2.1: Functional Requirements

S. No.	Functional Requirement	Type	Status

1	The user shall view product listings retrieved from multiple supported e-commerce platforms.	Core	Pending
2	The user shall view a section of "Trending Products" highlighted on the homepage.	Core	Pending
3	The user shall view real-time search suggestions while typing in the search bar.	Core	Pending
4	The user shall view a list of their recently viewed products for quick access.	Core	Pending
5	The user shall view a single unified product page containing prices from multiple vendors.	Intermediate	Pending
6	The user shall search for a product using keywords in the search bar.	Core	pending
7	The user shall view aggregated search results from multiple stores.	Core	Pending
8	The user shall sort search results by Price (Low to High).	Core	Pending

9	The user shall filter results by specific Store.	Core	Pending
10	The user shall filter results by Price Range (Min/Max).	Core	Pending
11	The user shall view detailed product information on a dedicated page.	Core	Pending
12	The user shall click "Open in Vendor Site" to navigate to the original vendor's website.	Core	Pending
13	The user shall create a new account (Sign up) using email and password.	Core	Pending
14	The user shall log into the system using valid credentials.	Core	Pending
15	The user shall log out of the system.	Core	Pending
16	The user shall recover a forgotten password via email.	Core	Pending

17	The user shall set a specific target price alert for a product.	Core	Pending
18	The user shall receive an email notification when a price drops to or below the target.	Core	Pending
19	The user shall receive a push notification on the mobile app for price drops.	Core	Pending
20	The user shall view a graphical history of price changes over time.	Core	Pending
21	The user shall add products to a personal "Wishlist".	Intermediate	Pending
22	The user shall remove items from their "Wishlist".	Intermediate	Pending
23	The user shall view updated prices refreshed at scheduled intervals.	Advanced	Pending
24	Admin shall log into the backend dashboard.	Backend	Pending
25	Admin shall view statistics on total products scraped and active users.	Intermediate	Pending
26	The user shall identify the best deals via automatically assigned "Best Value" badges.	Intermediate	Pending
27	The user shall view personalized product recommendations based on their browsing history.	Advanced	Pending

2.2. Non-Functional Requirements

A non-functional requirement is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviors. They are contrasted with functional requirements that define specific behaviors or functions.

iii: Table 2.2: Non-Functional Requirement

S. No.	Non-Functional Requirements	Category
1	The system interface shall be user-friendly and easy to navigate for new users.	Usability

2	The system shall ensure that user passwords and personal data are stored securely.	Security
3	The system shall provide helpful messages and feedback to user enters input.	Usability
4	The mobile application shall run smoothly on standard mobile devices.	Performance
5	The design shall use consistent fonts, colors, and icons throughout the application.	Usability
6	The website shall be responsive.	Usability

2.3. Selected Functional Requirements

Following is the list of the requirements selected for the current iteration.

iv: Table 2.3: Selected Functional Requirement

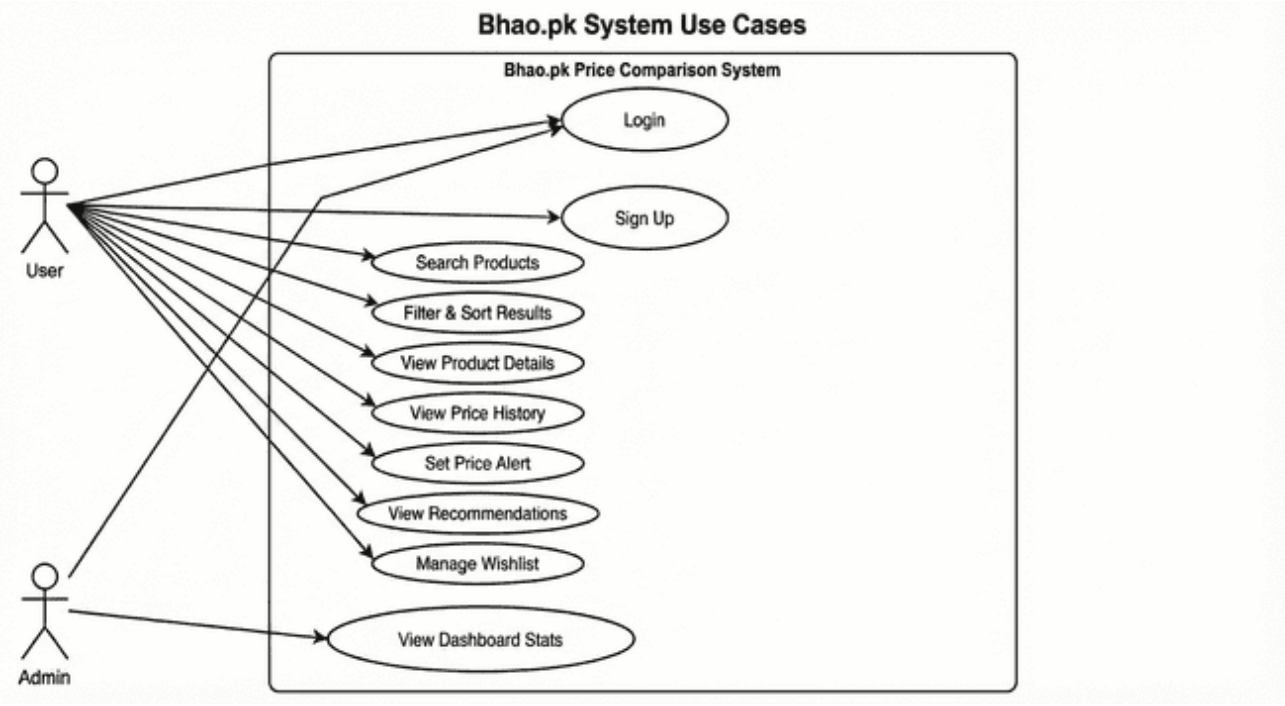
S. No.	Functional Requirement	Type
1	The user shall view product listings retrieved from multiple supported e-commerce platforms.	New
2	The user shall view a section of "Trending Products" highlighted on the homepage.	New
3	The user shall view real-time search suggestions while typing in the search bar.	New
4	The user shall view a list of their recently viewed products for quick access.	New
5	The user shall sort search results by Price (Low to High).	New

6	The user shall filter the results by specific Store.	New
7	The user shall filter the results by Price Range (Min/Max).	New
8	The user shall create a new account (Sign up) using email and password.	New
9	The user shall set a specific target price alert for a product.	New
10	The user shall log into the system using valid credentials.	New
11	The user shall log out of the system.	New
12	The user shall recover a forgotten password via email.	New
13	The user shall receive an email notification when a price drops to or below the target.	New
14	The user shall receive a push notification on the mobile app for price drops.	New
15	The user shall view a graphical history of price changes over time.	New
16	The user shall add products to a personal "Wishlist".	New
17	The user shall remove products from their "Wishlist".	
18	The user shall identify the best deals via automatically assigned "Best Value" badges.	New
19	The user shall view personalized product recommendations based on their browsing history.	New

20	Admin shall log into the backend dashboard.	New
21	Admin shall view statistics on total products scraped and active users.	New

2.4. System Use Case Modeling

A use case is a list of actions or event steps, typically defining the interactions between a role (known in the Unified Modeling Language as an actor) and a system, to achieve a goal. The actor can be a human or other external system. Customer's use cases are shown in the following the figure.



iii: Figure 2.1: Use case Diagram

v: Table 2.4: Search Product

Use Case ID:	Uc1

Use Case Name:	Search Product		
Created By:	Muhammad Ahad Ali Khan	Last Updated By:	Muhammad Haris Jamali
Date Created:	15/11/2025	Last Revision Date:	16/11/2025
Actors:	User (Guest or Registered)		
Description:	The user searches for a specific item to compare prices across stores.		
Trigger:	Search Button		
Preconditions:	The scrapper must be functional or the system must have scraped data available in the database.		
Postconditions:	A list of relevant products with prices is displayed.		
Normal Flow:	Customer	System	
	1: User enters keywords in search bar.	System queries the database using fuzzy matching.	
	2: User clicks search.	System displays aggregated results.	
Alternative Flows:	User applies filters (price/store) to the results.		
Exceptions:	1. No products found matching the query. 2. Database connection error.		

vi: Table 2.5: Set Price Alert

Use Case ID:	Uc2		
Use Case Name:	Set Price Alert		
Created By:	Muhammad Ahad Ali Khan	Last Updated By:	Muhammad Haris Jamali
Date Created:	15/11/2025	Last Revision Date:	16/11/2025
Actors:	Registered User		
Description:	User sets a target price to receive notifications when the product becomes cheaper.		
Trigger:	"Set Alert" Button		
Preconditions:	User must be logged in.		
Postconditions:	An alert record is created in the database.		
Normal Flow:	Customer	System	
	1: User views product details.	System verifies value is lower than current price.	
	2: User clicks "Set Alert".	System saves the alert preference.	
	3: User enters target price.		
Alternative Flows:	User hits back or cancel.		

Exceptions:	1. Target price is higher than the current price (System displays warning).

vii: Table 2.5: Set Price Alert

Use Case ID:	Uc3		
Use Case Name:	View Search Results		
Created By:	Muhammad Ahad Ali Khan	Last Updated By:	Muhammad Haris Jamali
Date Created:	15/11/2025	Last Revision Date:	16/11/2025
Actors:	User (Guest), Registered User		
Description:	The system presents a list of products that match the user's search query, showing key details like title, image, and price range.		
Trigger:	Successful completion of the "Search Products" use case.		
Preconditions:	A search query has been executed.		
Postconditions:	A curated list of product cards is presented to the user.		
Normal Flow:	Customer	System	
	1: User views search page.	System receives the list of matching products from the database.	
		System aggregates listings from different stores for the same product.	

		System displays products in a grid layout, showing image, title, and "Starts from Rs. X".
Alternative Flows:	1. The number of results exceeds one page limit (e.g., >20 items). 2. System displays pagination controls (e.g., "Next", "Page 2"). 3. User clicks "Next". 4. System fetches and displays the next set of results.	
Exceptions:	1. No results: System displays "No products found."	

viii: Table 2.7: View Product Details

Use Case ID:	Uc4		
Use Case Name:	View Product Details		
Created By:	Muhammad Ahad Ali Khan	Last Updated By:	Sheryar Karim
Date Created:	15/11/2025	Last Revision Date:	16/11/2025
Actors:	User (Guest), Registered User		
Description:	The user views full details and a price comparison table for a product.		
Trigger:	User clicks a product card.		
Preconditions:	Product ID exists.		
Postconditions:	Product Detail Page is displayed.		
Normal Flow:	Customer		System

	1: User clicks a product.	System receives the list of matching products from the database.
	2: User sees product details, store listings, and history.	System aggregates listings from different stores for the same product.
	3: User can proceed to other actions.	System displays products in a grid layout, showing image, title, and "Starts from Rs. X".
Alternative Flows:	User decides to go to some other page from navigation.	
Exceptions:	1. Product not found: Redirect to 404 page.	

ix: Table 2.8: Filter & Sort Results

Use Case ID:	Uc5		
Use Case Name:	Filter & Sort Results		
Created By:	Muhammad Ahad Ali Khan	Last Updated By:	Muhammad Haris Jamali
Date Created:	15/11/2025	Last Revision Date:	16/11/2025
Actors:	User (Guest), Registered User		
Description:	The user applies criteria (price range, store) or changes the order (low-to-high) to narrow down search results.		
Trigger:	User interacts with filter sidebar or sort dropdown.		

Preconditions:	Search results must be currently displayed.	
Postconditions:	The product list is updated to reflect specific criteria.	
Normal Flow:	Customer	System
	1. User selects a filter (e.g., Store: Daraz) or sort order.	System filters the current list.
		System filters the current list.
		System refreshes the grid with updated results.
		System refreshes the grid with updated results.
Alternative Flows:	User clicks "Clear All" to revert to the original search results.	
Exceptions:	System displays "No products match your filters."	

x: Table 2.9: View Price History

Use Case ID:	Uc6		
Use Case Name:	View Price History		
Created By:	Muhammad Ahad Ali Khan	Last Updated By:	Muhammad Haris Jamali
Date Created:	15/11/2025	Last Revision Date:	16/11/2025
Actors:	User (Guest), Registered User		
Description:	User views a graph of price changes over time.		
Trigger:	User views Product Detail Page.		
Preconditions:	Historical price data exists.		
Postconditions:	Price graph is displayed.		
Normal Flow:	Customer	System	
	1: User loads product page.	System fetches historical data.	
		System renders graph.	
Alternative Flows:	1. User changes graph view from default "30 Days" to "6 Months" via a control. 2. System fetches additional historical data from the database. 3. System re-renders the graph with the new time scale.		
Exceptions:	Insufficient data**:** System doesn't show the graph and instead says it doesn't have enough historical data to construct a graph.		

xi: Table 2.10: Sign Up

Use Case ID:	Uc7		
Use Case Name:	Sign Up		
Created By:	Muhammad Ahad Ali Khan	Last Updated By:	Muhammad Ahad Ali Khan
Date Created:	15/11/2025	Last Revision Date:	21/11/2025
Actors:	User (Guest)		
Description:	New user creates an account.		
Trigger:	User clicks "Sign Up".		
Preconditions:	User email is not already registered.		
Postconditions:	New user account created and logged in.		
Normal Flow:	Customer	System	
	1: User provides email and password.	System validates and checks uniqueness.	
	2:: User clicks create account.	System creates user record.	
		System logs user in.	
Alternative Flows:	User clicks login link from the signup page.		

Exceptions:	Email exists: Show error message.

xii: Table 2.11: Login

Use Case ID:	Uc8		
Use Case Name:	Login		
Created By:	Muhammad Haris Jamali	Last Updated By:	Sheryar Karim
Date Created:	18/11/2025	Last Revision Date:	21/11/2025
Actors:	User (Guest), Admin		
Description:	Existing user authenticates.		
Trigger:	User clicks "Login".		
Preconditions:	User/Admin has an account.		
Postconditions:	User is authenticated as Registered User. Admin is authenticated as Admin		
Normal Flow:	Customer	System	
	1: User/Admin enters email and password. .	System verifies credentials.	
	2: User/Admin clicks create account.	System creates session.	
Alternative Flows:	User clicks "Forgot Password?" link on login form.		

Exceptions:	Invalid credentials: Show error message.

xiii: Table 2.12: View Recommendations

Use Case ID:	Uc9		
Use Case Name:	View Recommendations		
Created By:	Muhammad Haris Jamali	Last Updated By:	Muhammad Ahad Ali Khan
Date Created:	18/11/2025	Last Revision Date:	23/11/2025
Actors:	Registered User		
Description:	User sees personalized product suggestions.		
Trigger:	User visits homepage.		
Preconditions:	User is logged in with some activity history.		
Postconditions:	Recommended items are displayed.		
Normal Flow:	Customer	System	
	.	System identifies user.	
		System retrieves user persona and finds matching products.	

		System displays recommendations.
Alternative Flows:	User clicks a "Show More" button in the recommendations section.	
Exceptions:	No history: System shows popular items.	

xiv: Table 2.13: Manage Wishlist

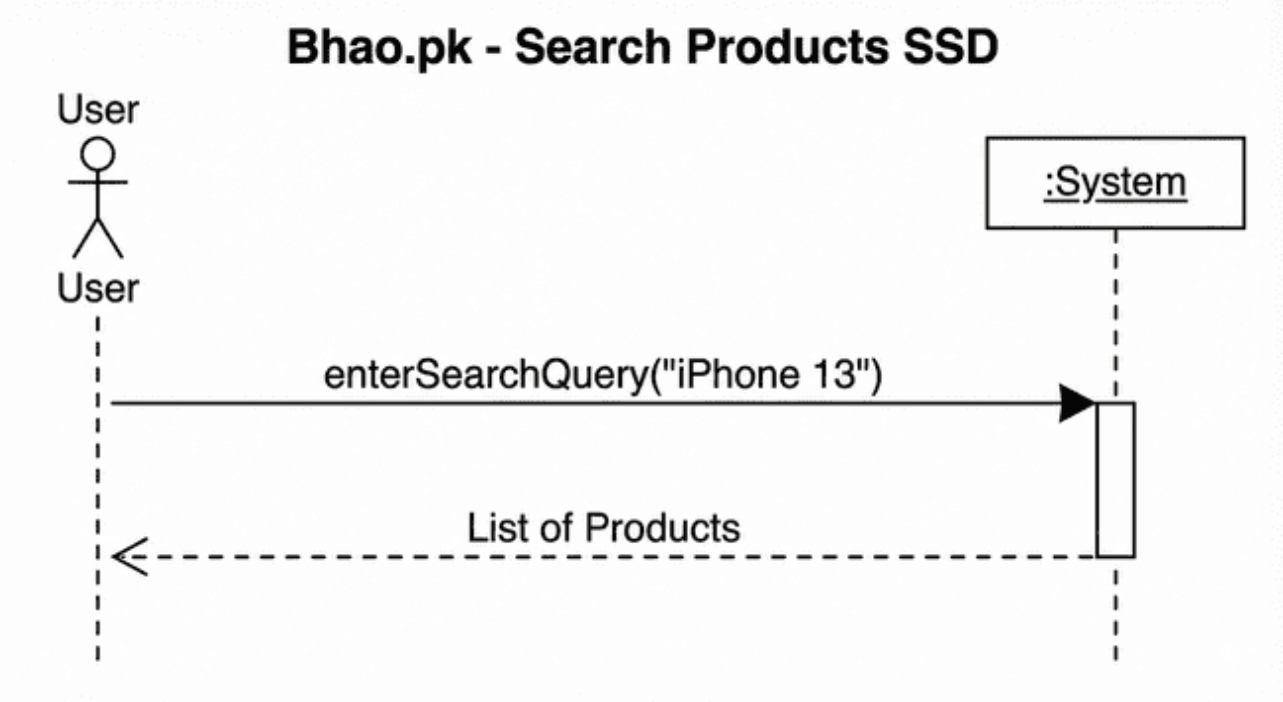
Use Case ID:	Uc10		
Use Case Name:	Manage Wishlist		
Created By:	Sheryar Karim	Last Updated By:	Muhammad Ahad Ali Khan
Date Created:	20/11/2025	Last Revision Date:	24/11/2025
Actors:	Registered User		
Description:	The user saves products to a personal list for future reference or removes them when no longer needed.		
Trigger:	User clicks the "Heart" icon or "Add to Wishlist" button.		
Preconditions:	User is logged in.		
Postconditions:	Product is added to or removed from the user's saved list in the database		
Normal Flow:	Customer		System

	1: User clicks the "Heart" icon on a product card.	System checks if the item is already in the wishlist.
		System adds the item ID to the user's wishlist record.
		System updates the icon to appear "filled".
Alternative Flows:	If the item is already in the wishlist, clicking the icon removes it.	
Exceptions:	System fails to save the item and displays an error "Try again later".	

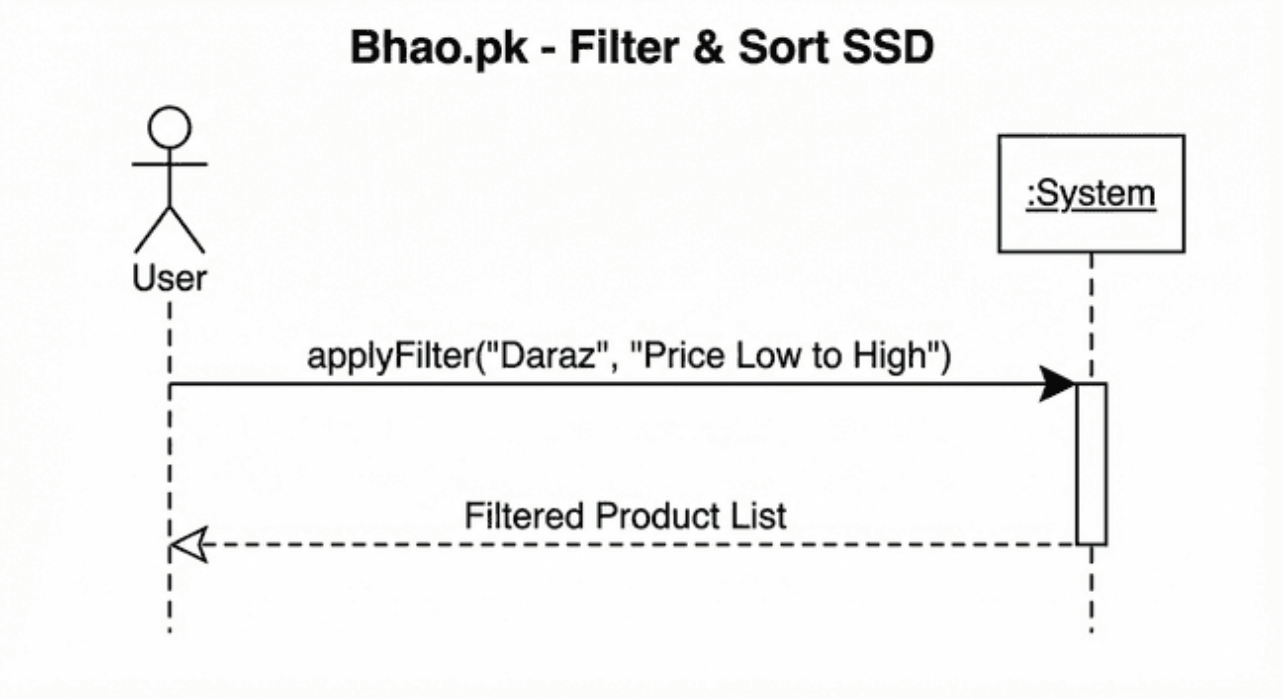
xv: Table 2.14: View Dashboard Stats

Use Case ID:	Uc11		
Use Case Name:	View Dashboard Stats		
Created By:	Sheryar Karim	Last Updated By:	Muhammad Haris Jamali
Date Created:	20/11/2025	Last Revision Date:	24/11/2025
Actors:	Admin		
Description:	The admin views system health metrics, including total products scraped and active user counts.		
Trigger:	Admin logs into the backend portal.		

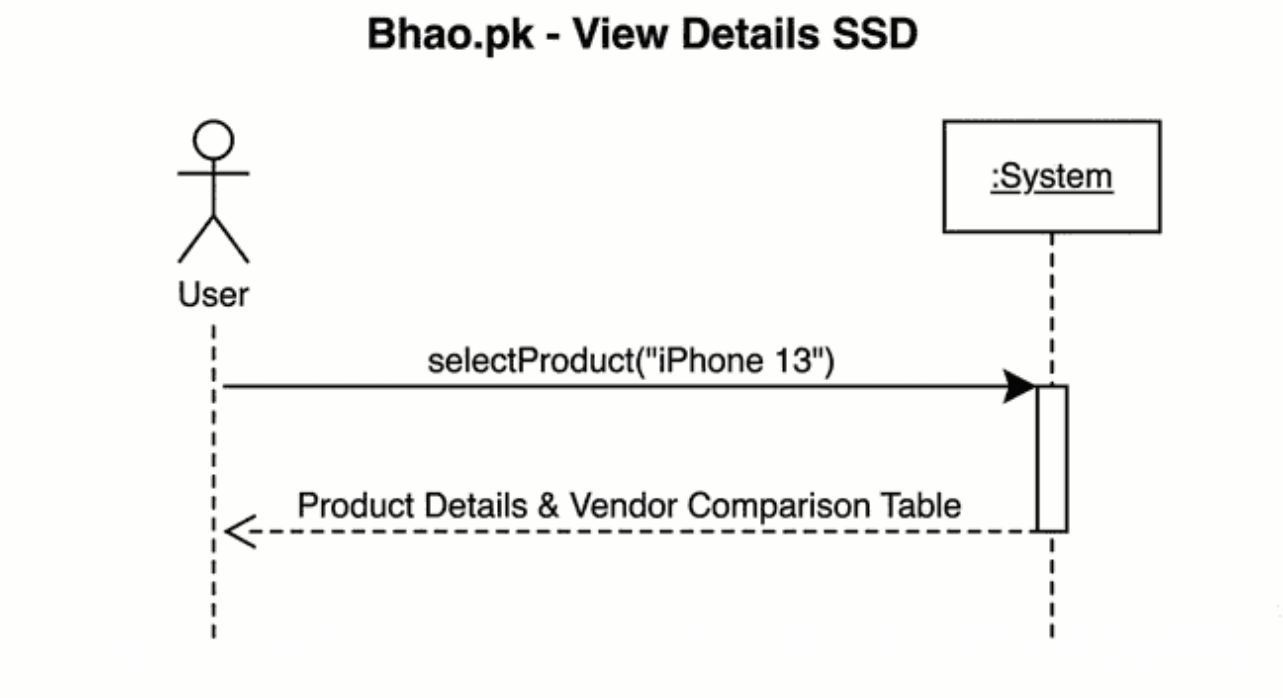
Preconditions:	Admin must be authenticated.	
Postconditions:	Statistical graphs and counters are displayed.	
Normal Flow:	Admin	System
	1: Admin navigates to the main Dashboard tab.	System queries the database for total product count and user logs.
		System renders charts (e.g., Scraper Status, New Signups).
Alternative Flows:	None.	
Exceptions:	System displays 0 values if the database is empty.	



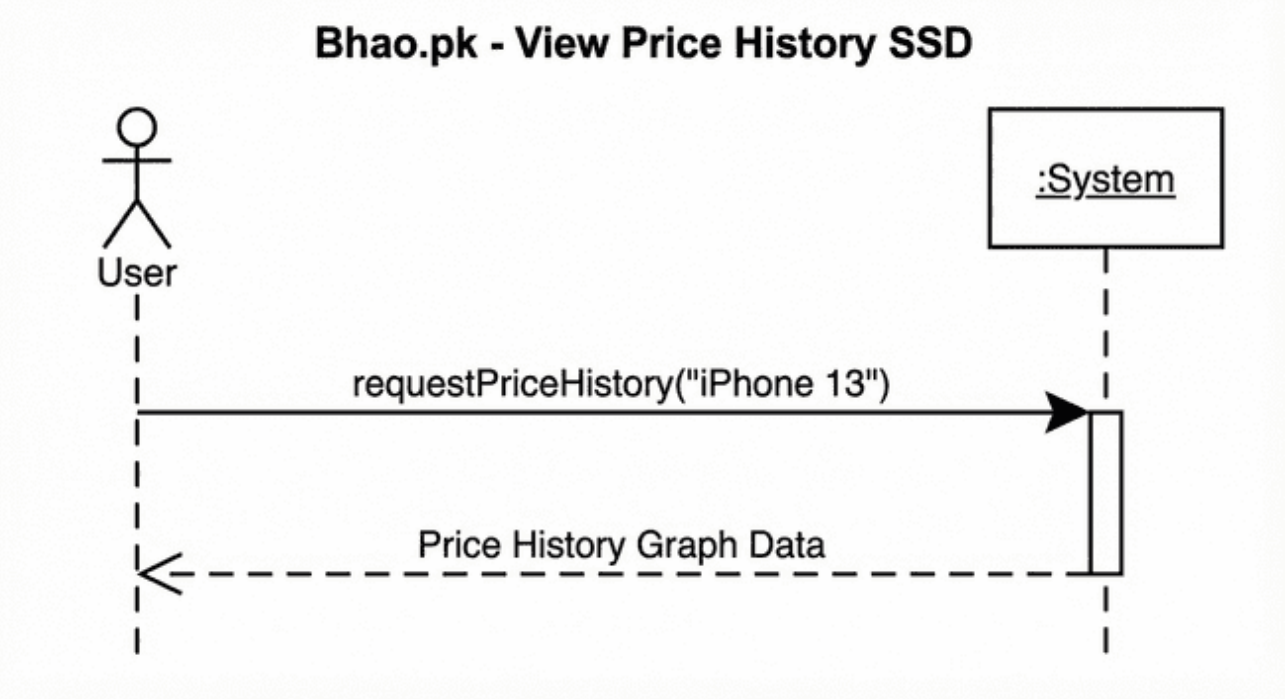
iv: Figure 2.2: SSD Search Products SSD



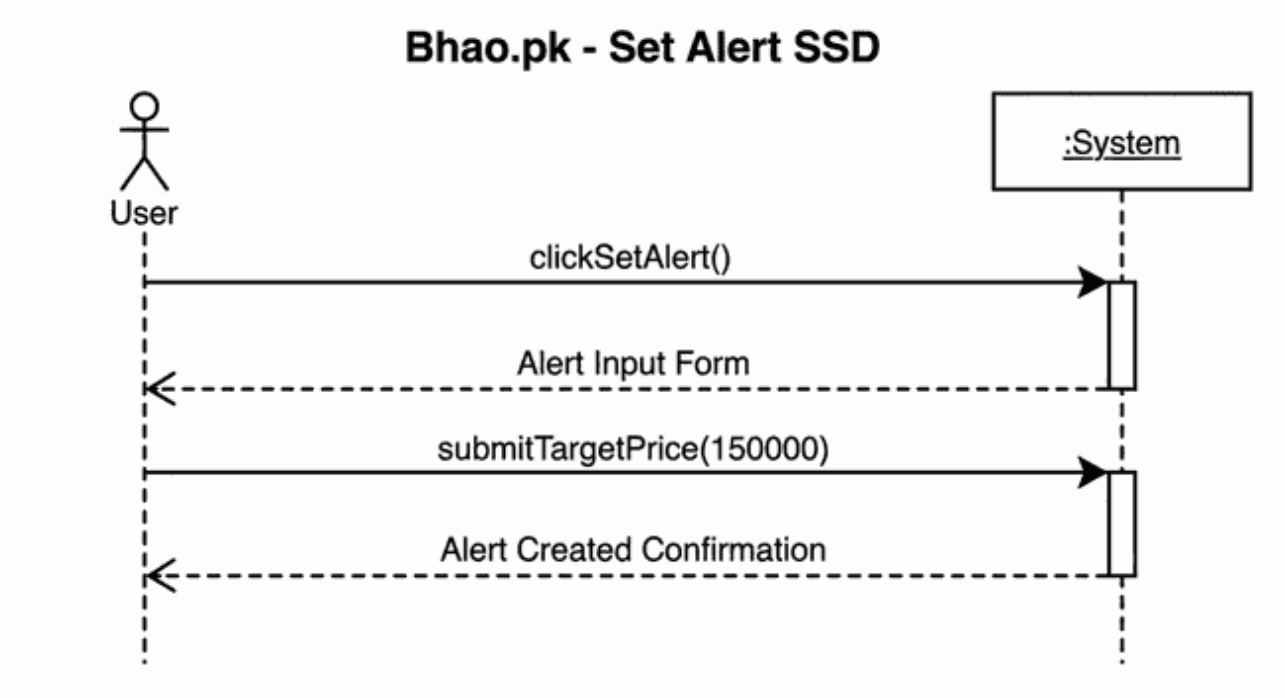
v: Figure 2.3: SSD Filter and Sort SSD



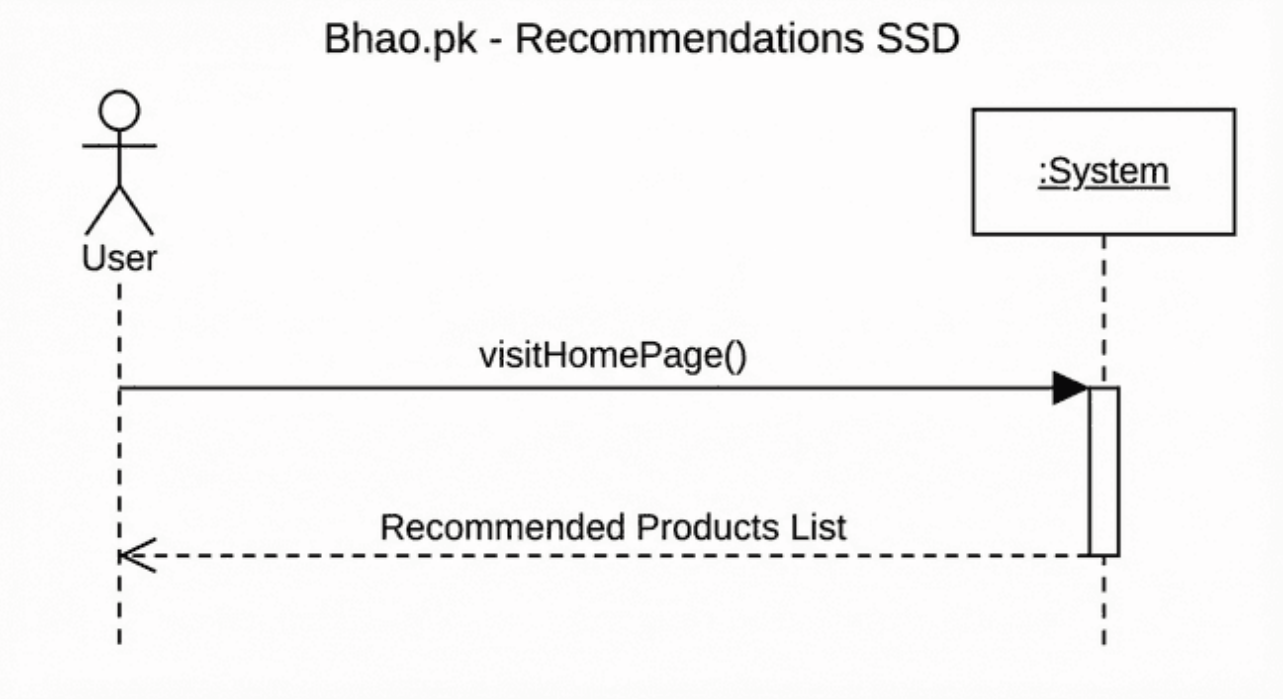
vi: Figure 2.4: View Details SSD



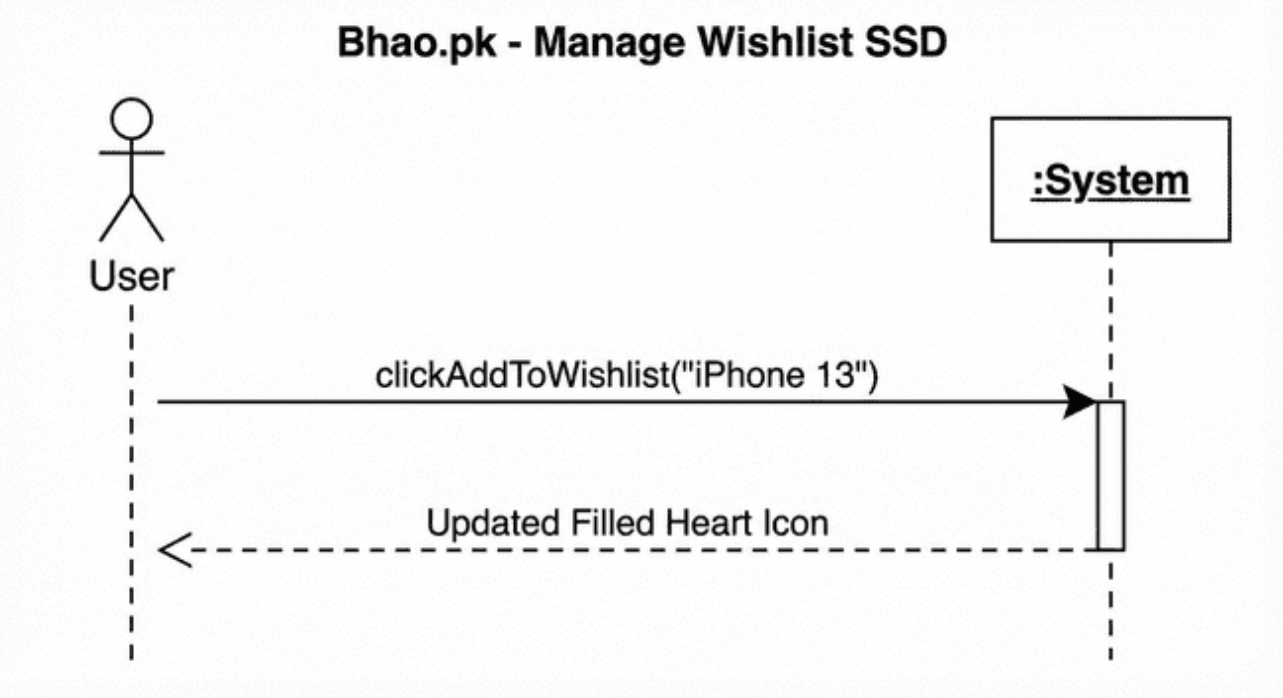
vii: Figure 2.5: SSD View Price History SSD



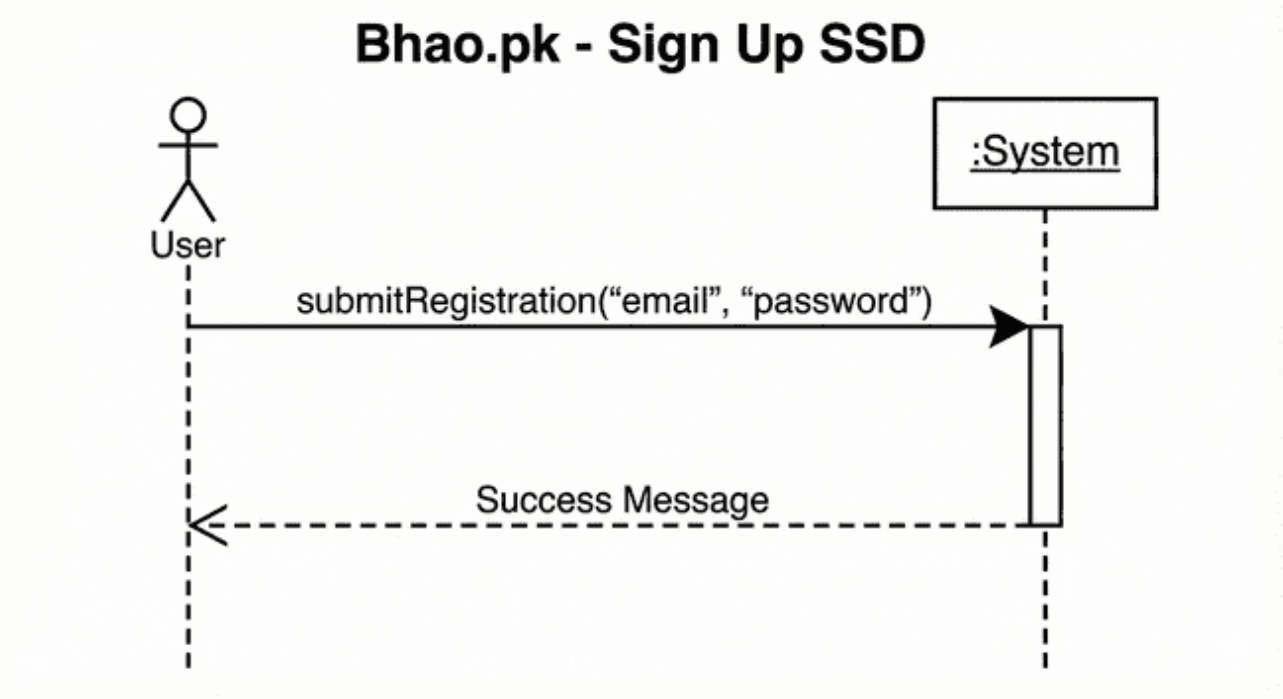
viii: Figure 2.6: Set Price Alert SSD



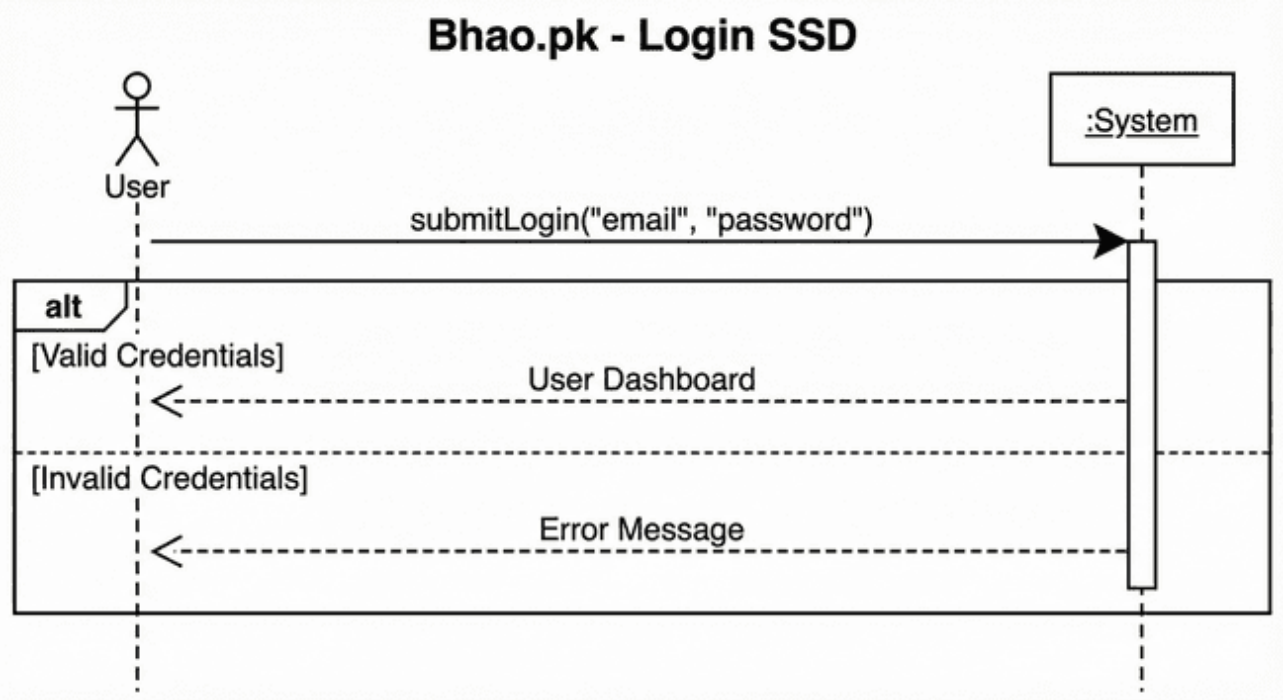
ix: Figure 2.7: Product Recommendation SSD



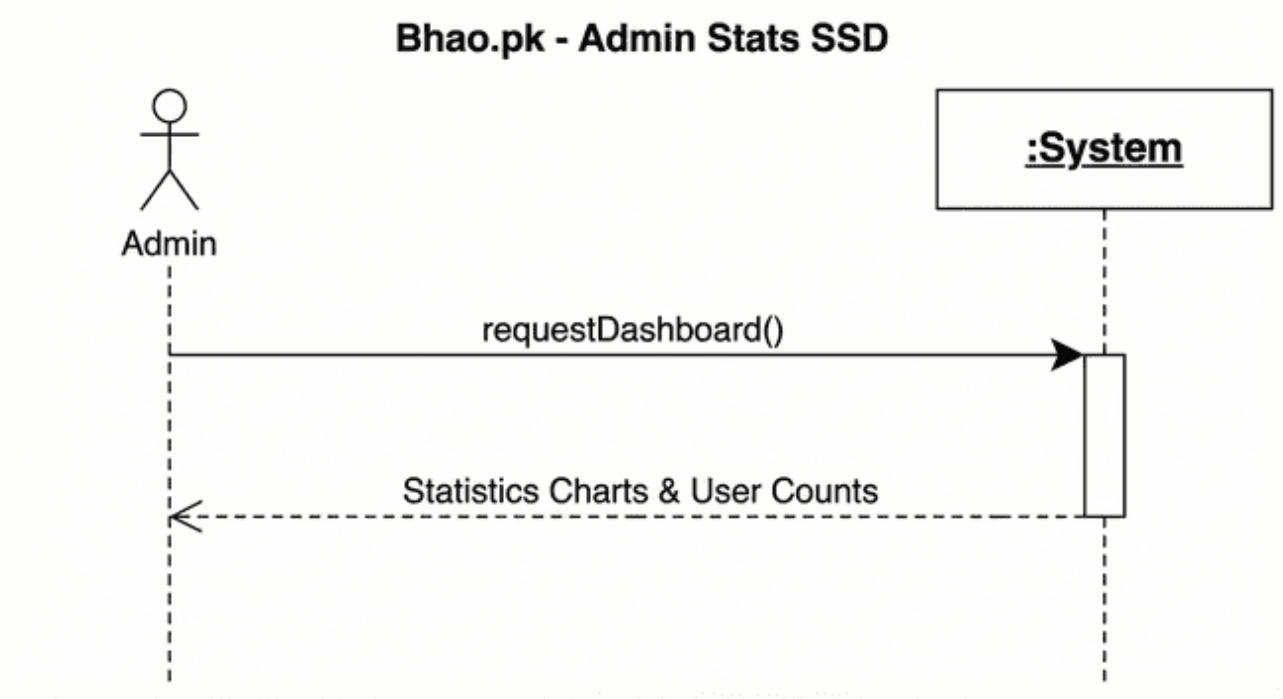
x: Figure2.8: Manage Wishlist SSD



xi: Figure 2.9: Sign Up SSD



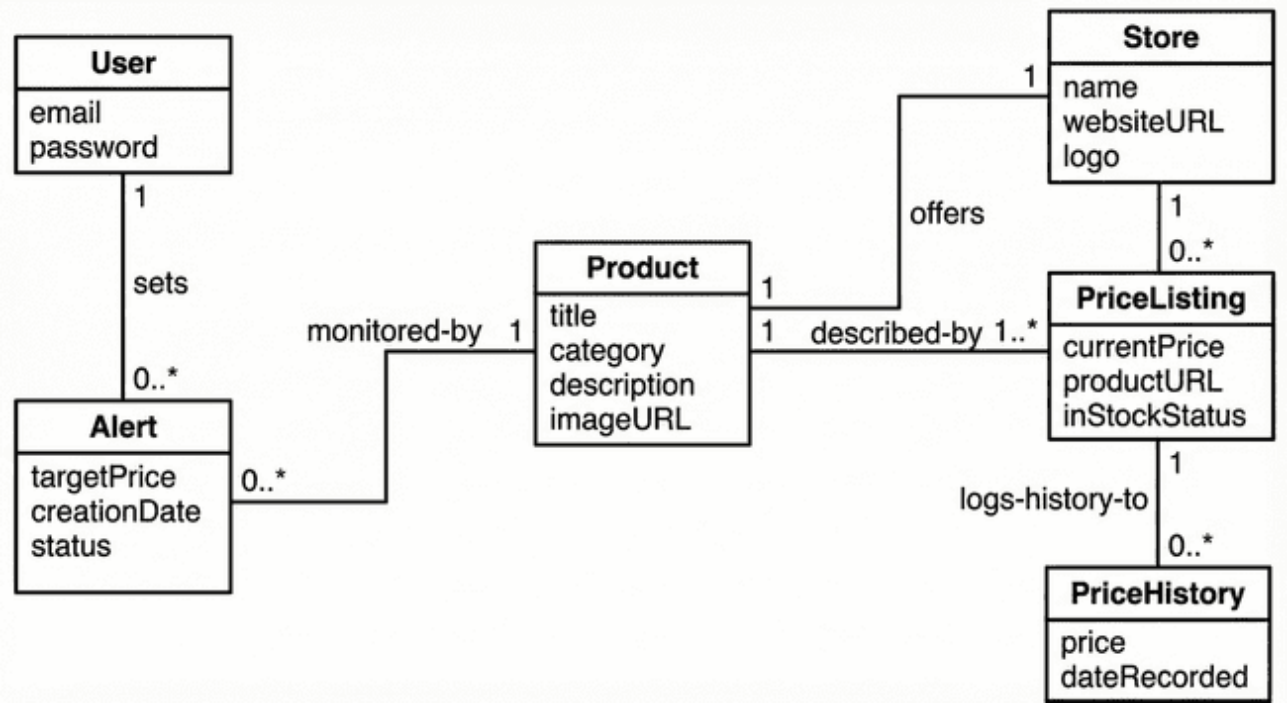
xii: Figure 2.10: Login SSD



xiii: Figure 2.11: Admin Stats SSD

2.6. Domain Model

The Domain Model for Bhao.pk illustrates the conceptual structure of the price comparison ecosystem, centering on the **Product** entity as the core item being tracked. Each generic product is linked to multiple **StoreListings**, representing specific offers from external **Stores** like Daraz or Telemart. **Users** interact with the system by setting **Alerts** on these products to monitor price changes. To maintain up-to-date information, a backend **ScraperBot** creates and updates listings, while a **PriceHistory** entity logs every price fluctuation over time to support historical trend analysis.



xiv: Figure 2.19: Domain Model

Chapter 3

System Design

The system design of Bhao.pk is built upon a robust, multi-tiered **Software Architecture** that ensures scalability and clear separation of concerns. It is organized into three primary layers: the **Presentation Layer**, which handles user interactions via a React-based frontend; the **Business Logic Layer**, which serves as the core engine managing APIs, the automated **Scraping Manager**, and the Alert System; and the **Database Layer**, which ensures persistent data storage using PostgreSQL

3.1. Layer Definition

xvii: Table 3.1: Layers Definition

Layers	Description
Presentation Layer	This layer will be used for the interaction with the user through a graphical user interface.
Business Logic Layer	This layer contains the business logic. All the constraints and majority of the functions reside under this layer.
Database Layer	This layer contains the database of the application being developed.

3.1.1. Presentation Layer:

This layer is responsible for the user interface and user experience. It handles user inputs and displays data retrieved from the server. It includes the web frontend (React.js) and potential future mobile interfaces.

3.1.2. Business Logic Layer:

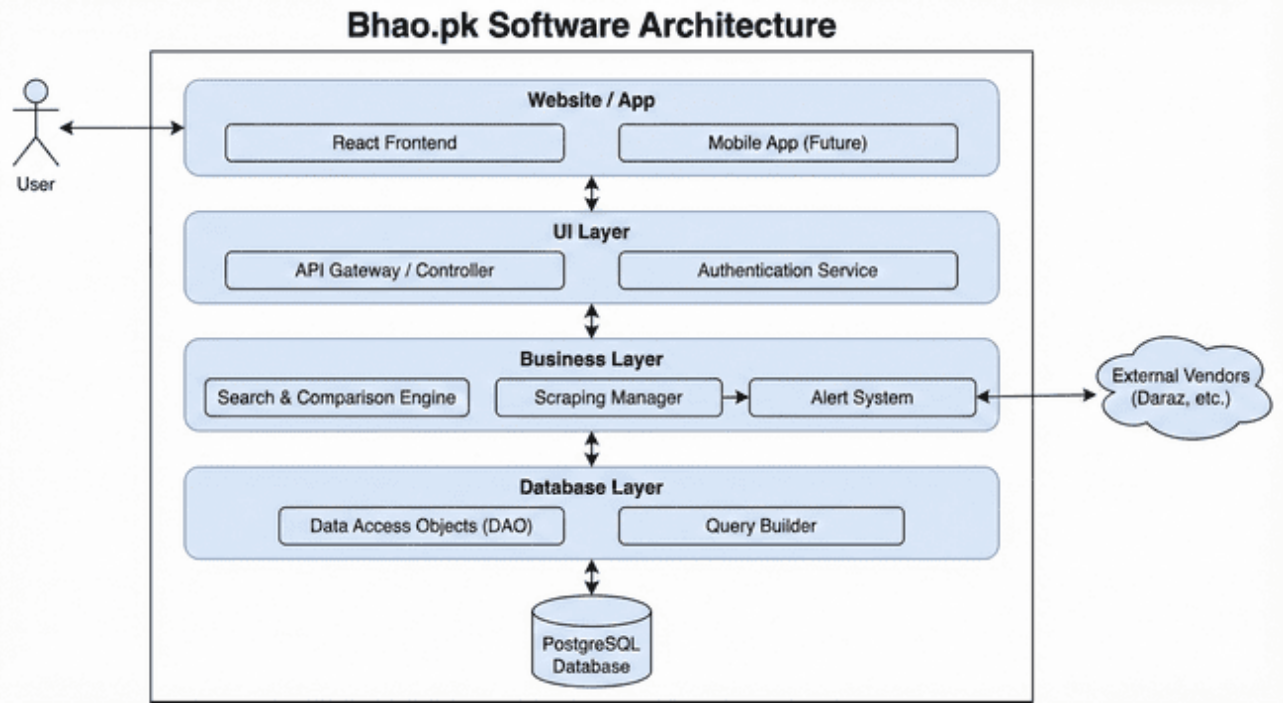
This layer contains the core functionality. It manages data processing, executes search algorithms, handles scraping scheduling, and processes alerts. It serves as the bridge between the UI and the database.

3.1.3. Database Layer:

Database layer includes database servers where information is stored and retrieved. Data in this tier is kept independent of application servers or business logic.

3.2. Software Architecture

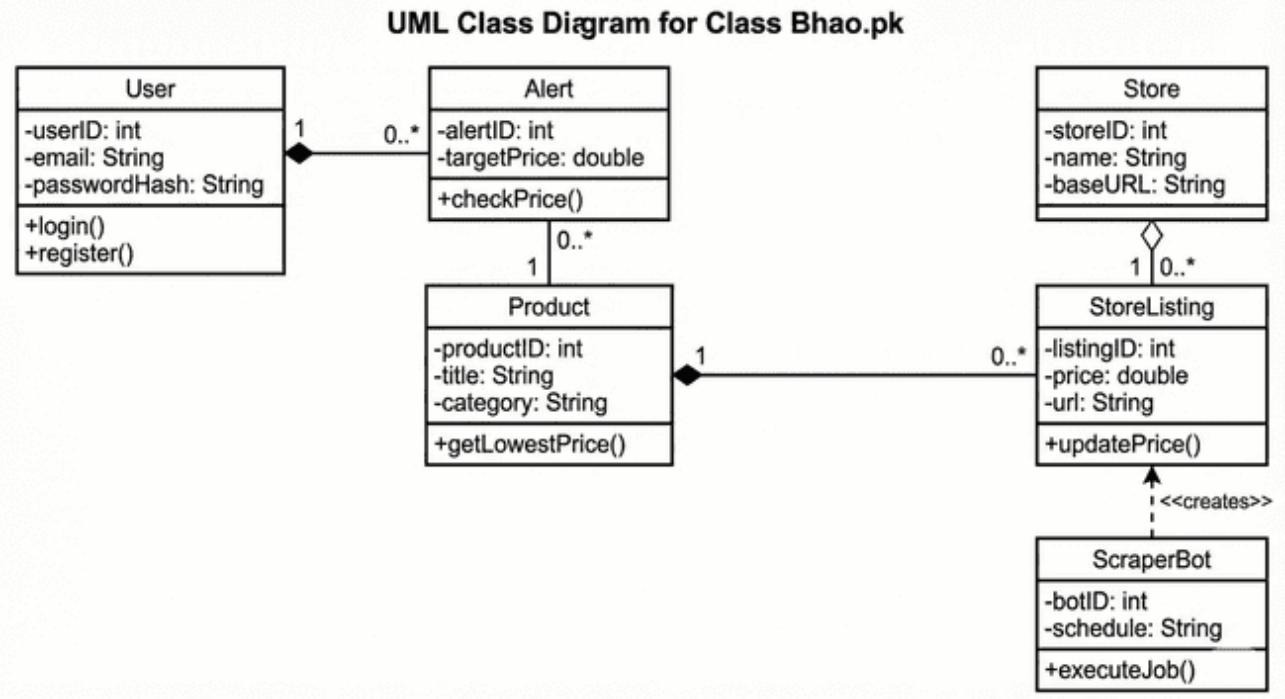
This layer is responsible for persistent data storage. It manages the relational database (PostgreSQL) holding information on users, products, stores, and historical price records. Below is the architecture diagram of the system:



xv: Figure 3.1: Software Architecture Diagram

3.3. Class Diagram

The Class Diagram for Bhao.pk defines the system's static structure, centering on the **Product** class which is composed of multiple specific **StoreListings** to represent offers from different vendors. **Stores** act as aggregators for these listings, while **Users** maintain a strict composition relationship with **Alerts**, allowing for personalized price monitoring that depends on the user account's existence. Additionally, the diagram illustrates backend automation through the **ScraperBot** class, which demonstrates a dependency on StoreListings to dynamically create and update price data from external sources.



xvi: Figure 3.2: Class Diagram

User:

This class represents a user of the Bhao.pk system. It stores essential account information such as userId, email, passwordHash, and their role (e.g., guest or registered). The main responsibilities of this class include managing user authentication via login() and creating new accounts via register().

Alert:

This class represents a price notification set by a registered user. It holds data including its unique alertId, the user's desired targetPrice, the creationDate, and a boolean isActive status. Its primary operation, checkPrice(), compares a product's current price against the target price to determine if a notification should be triggered.

Product:

This is the central entity of the system, representing a generic item that can be sold by multiple vendors. It contains general attributes like productId, title, category, and a baseImage URL. Its key operations include getLowestPrice() to calculate the best available deal and getListings() to retrieve all associated vendor offers.

StoreListing:

This class represents a specific offer for a Product from a single Store. It links the general product to a concrete vendor and contains listing-specific details like listingId, current price, the direct vendorUrl to the purchase page, and a lastUpdated timestamp. It includes an updatePrice() operation to modify its price based on new data.

Store:

This class represents an external vendor or e-commerce website (e.g., Daraz, PriceOye). It stores identity information such as the storeId, the store's name, its baseUrl, and a logoUrl.

ScraperBot:

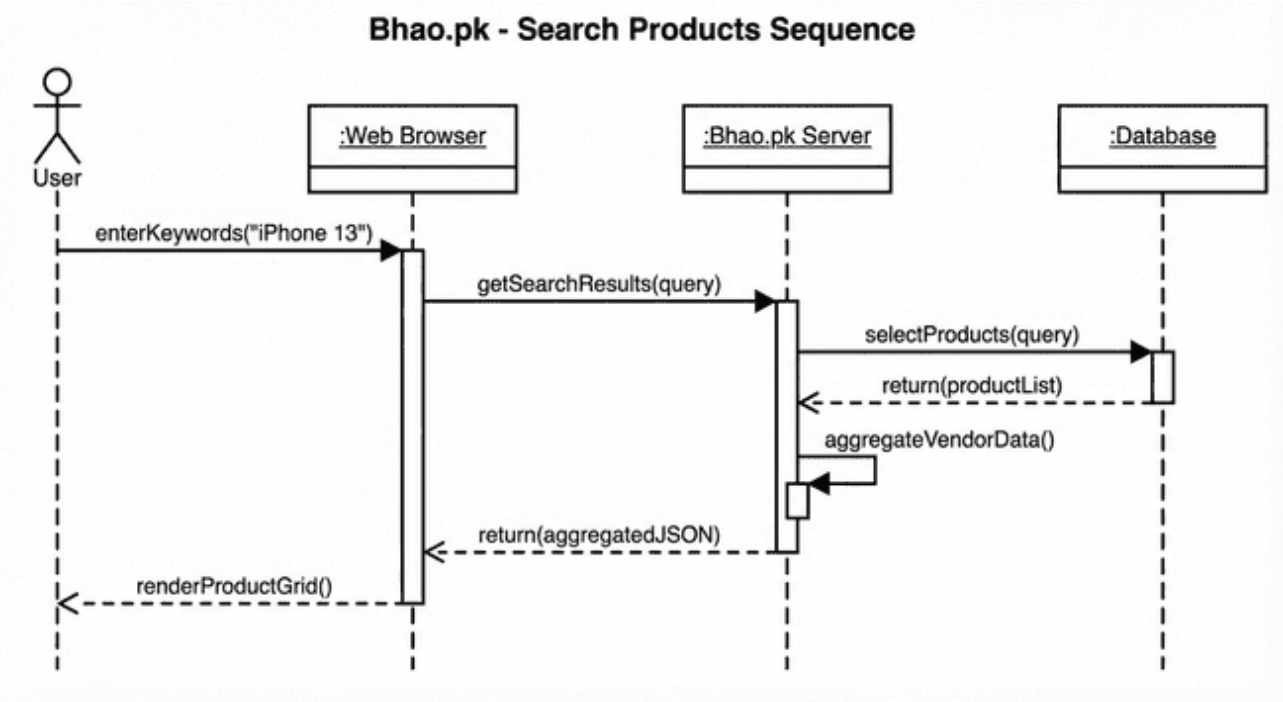
This class is part of the backend system responsible for automated data extraction. It includes attributes for its botId, the targetDomain it is assigned to crawl, and its execution schedule. Its operations include executeJob() to start the crawling process and extractData() to parse product information from raw HTML.

3.4. Sequence Diagram

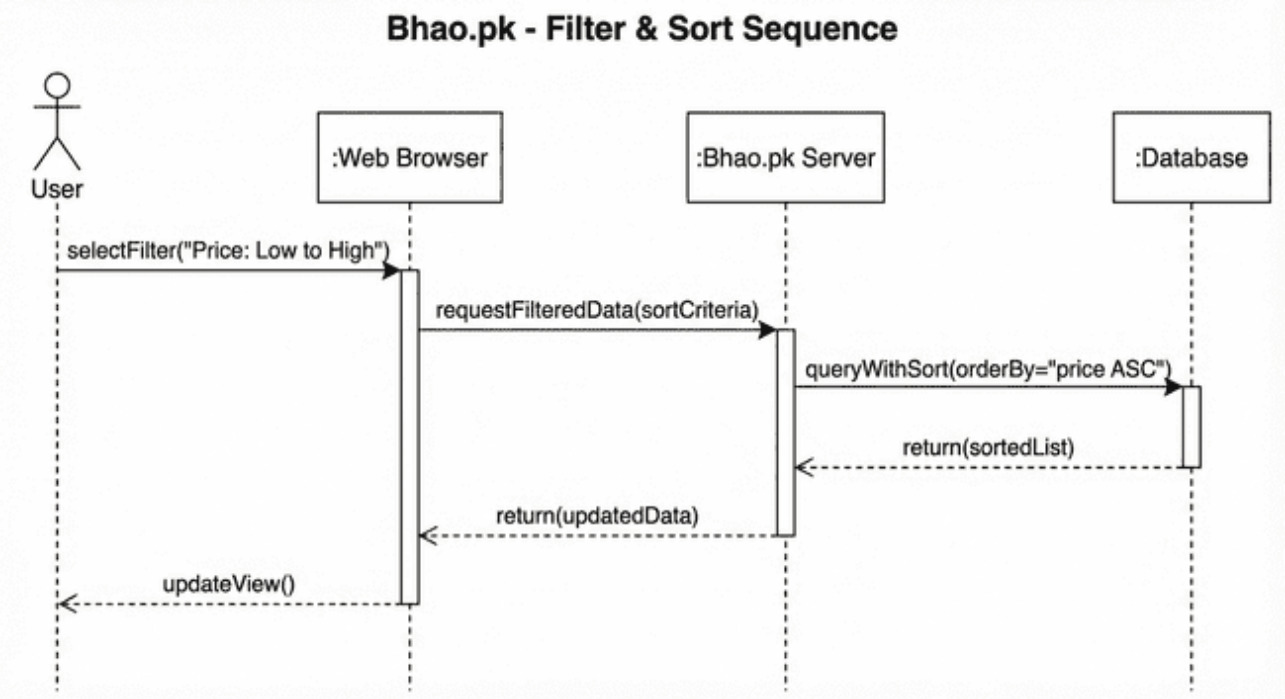
The Sequence Diagrams for Bhao.pk model the dynamic behavior of the system, illustrating the precise step-by-step message flows between the User, Web Browser, Server, and Database. They provide a detailed blueprint of the logic required to execute core functionalities, such as Searching for Products, Viewing Price History, and Setting Alerts, by visualizing how frontend API requests are processed, validated, and responded to by the backend architecture.

3.4.1. User

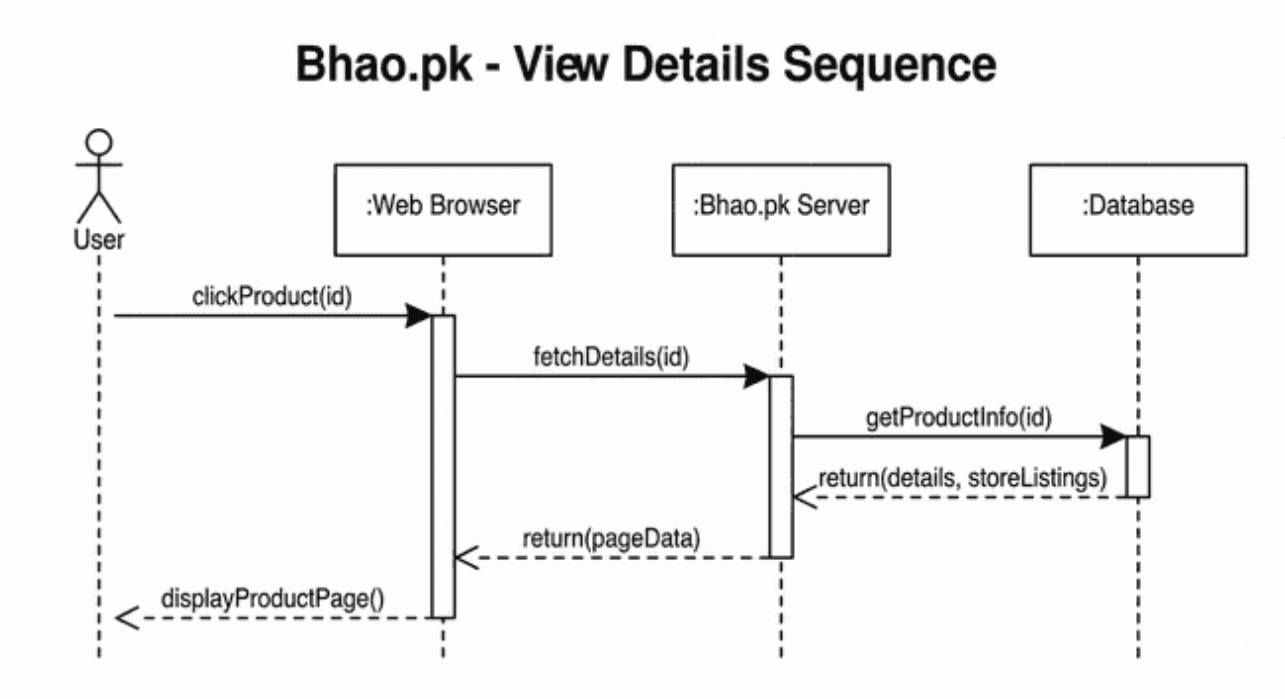
This Sequence diagram shows the flow of User actions how he interacts with the system and how he performs general tasks like sign up and login.



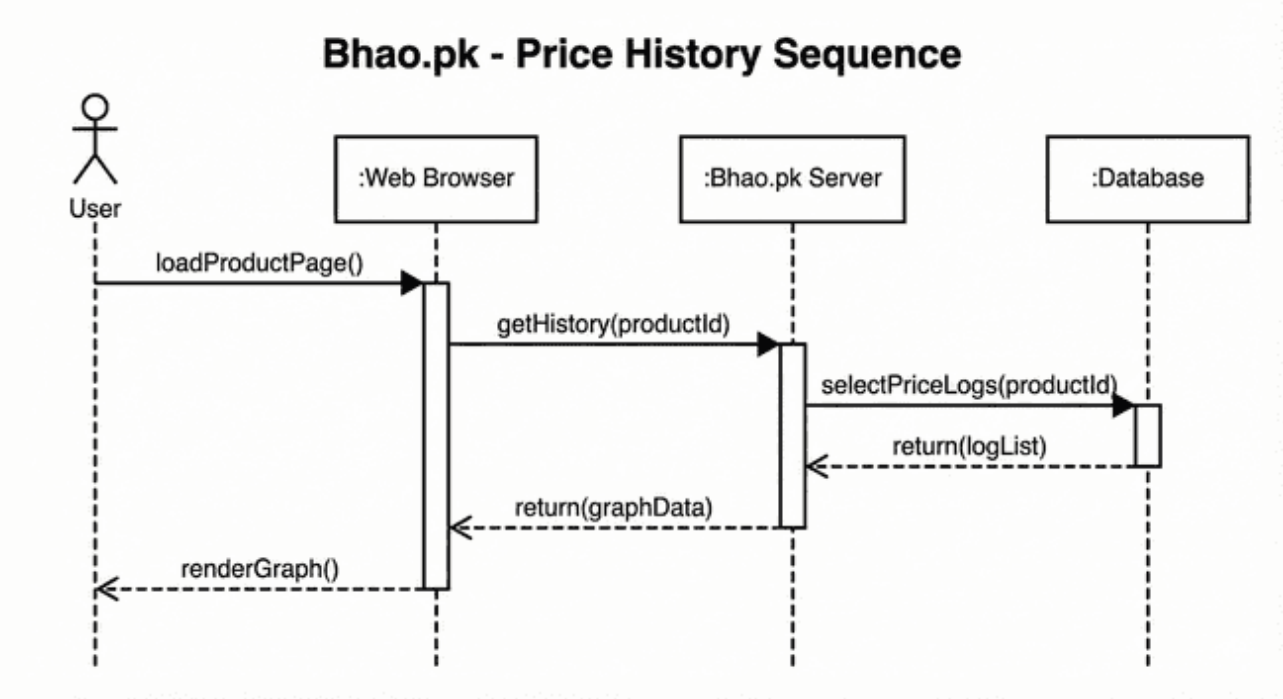
xvii: Figure 3.2: Search Products Sequence Diagram



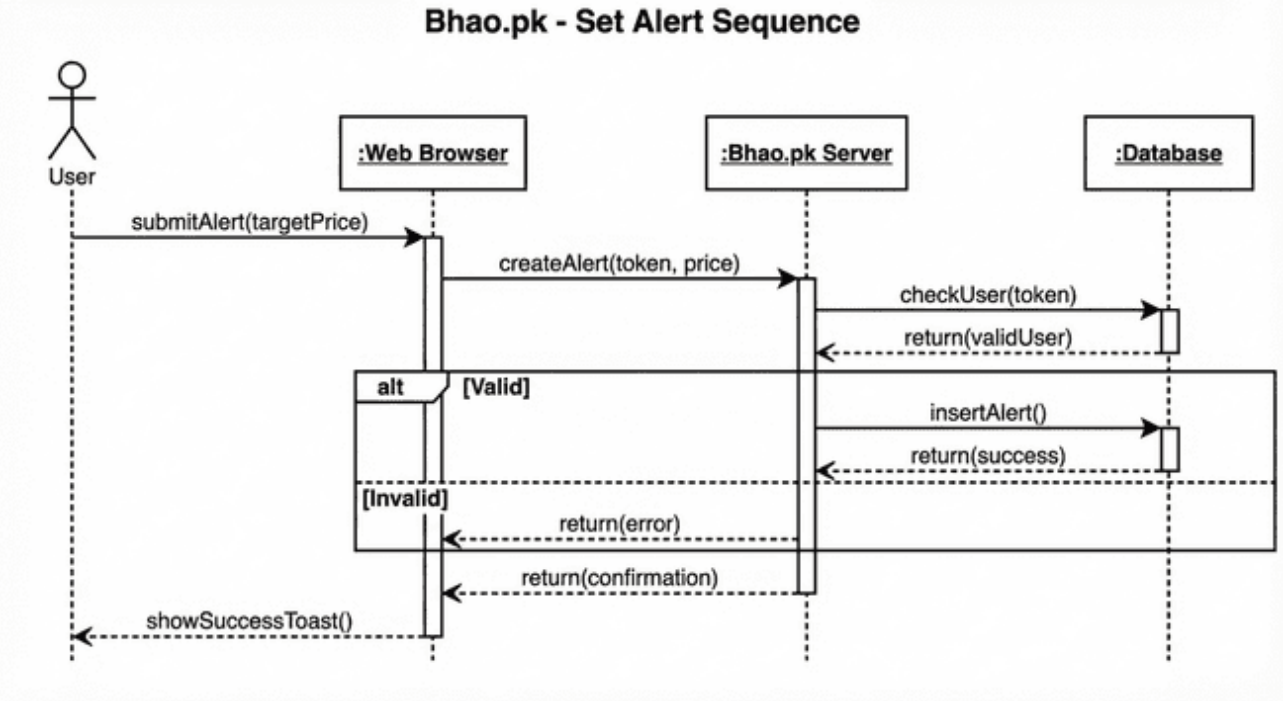
xviii: Figure 3.3: Filter & Sort Sequence Diagram



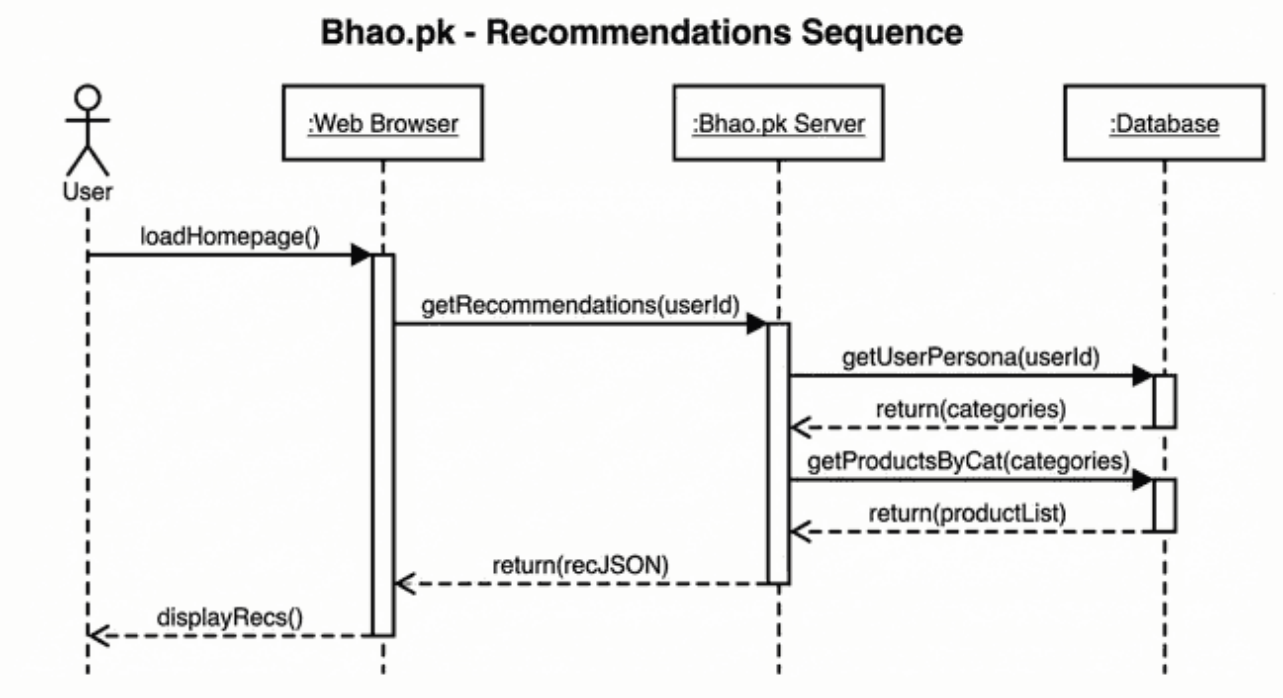
xix: Figure 3.4: View Details Sequence Diagram



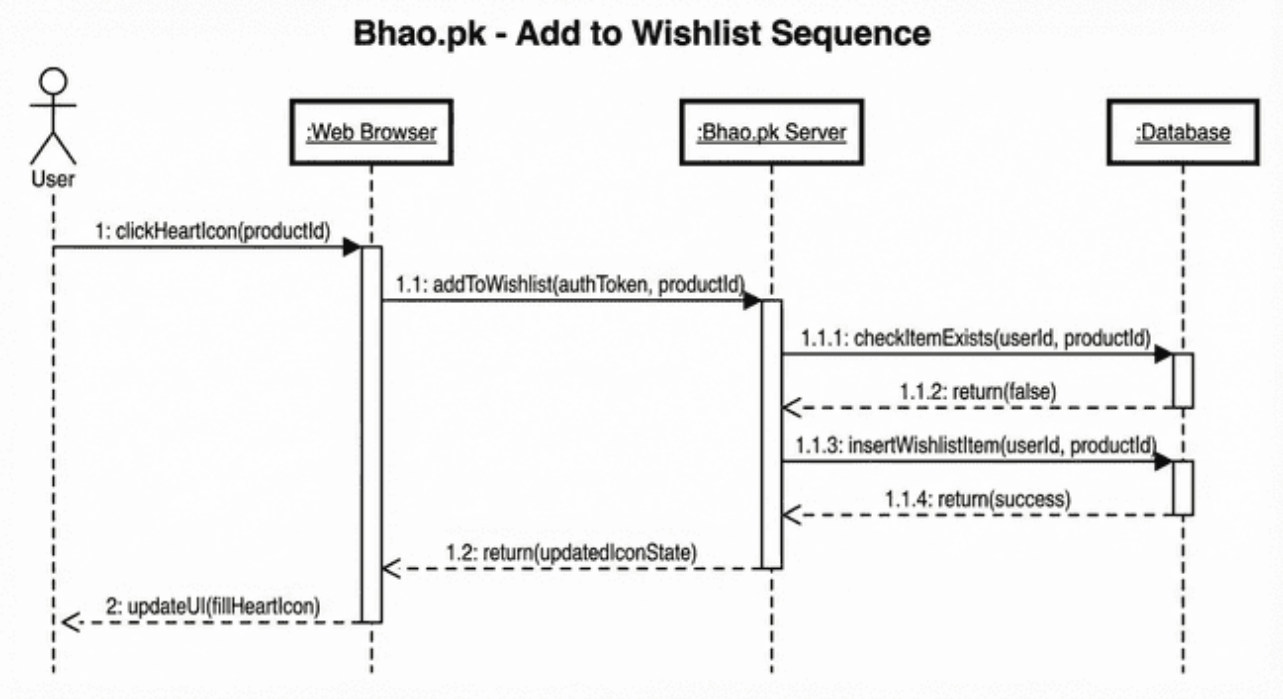
xx: Figure 3.5: Price History Sequence Diagram



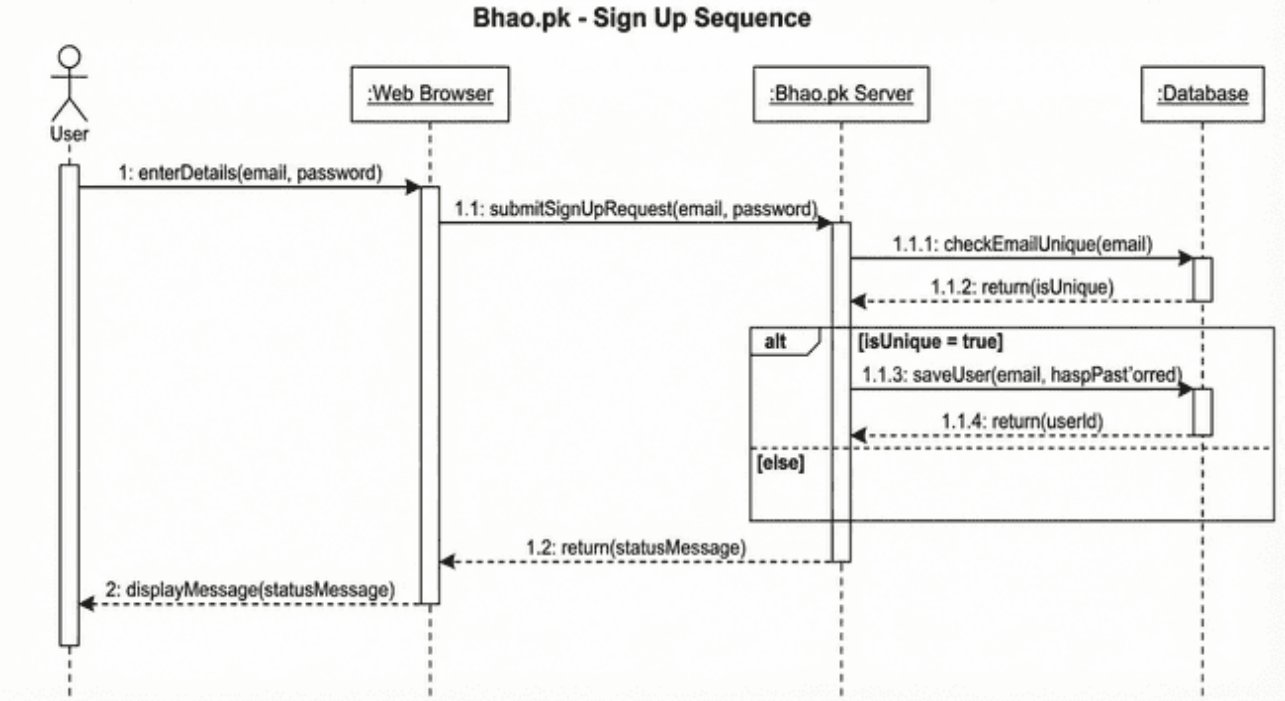
xxi: Figure 3.6: Set Price Alert Sequence Diagram



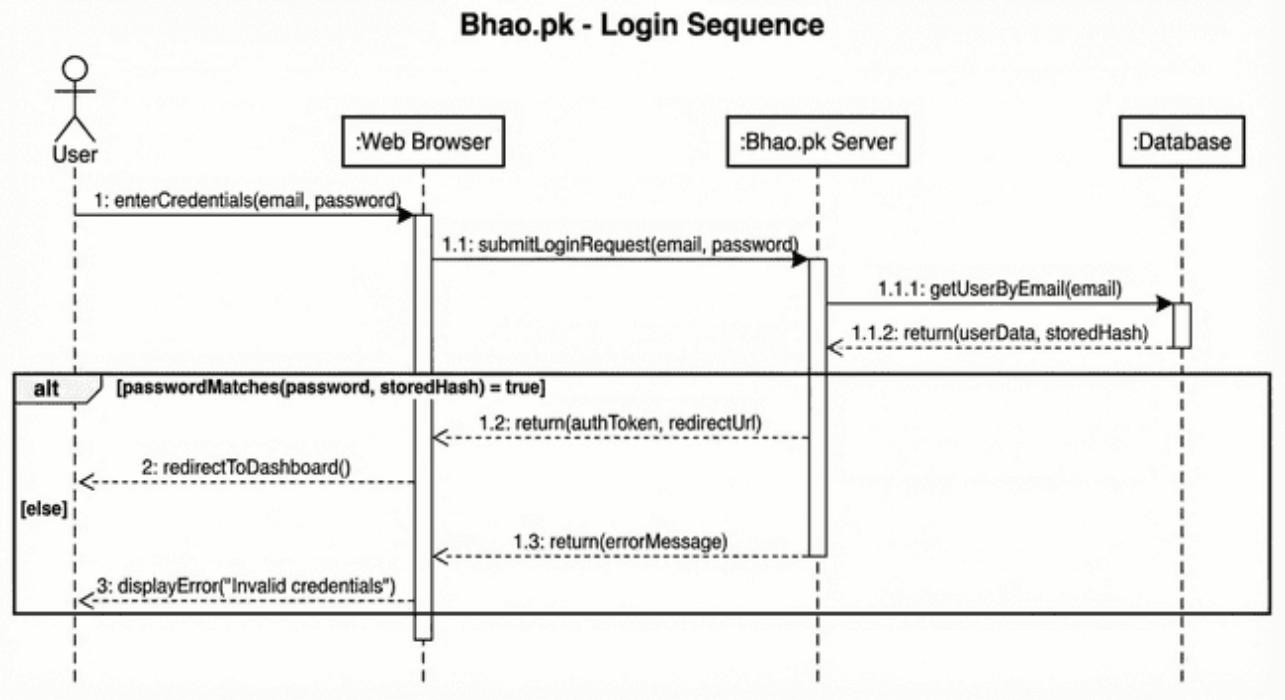
xxii: Figure 3.7: Show Recommendations Sequence Diagram



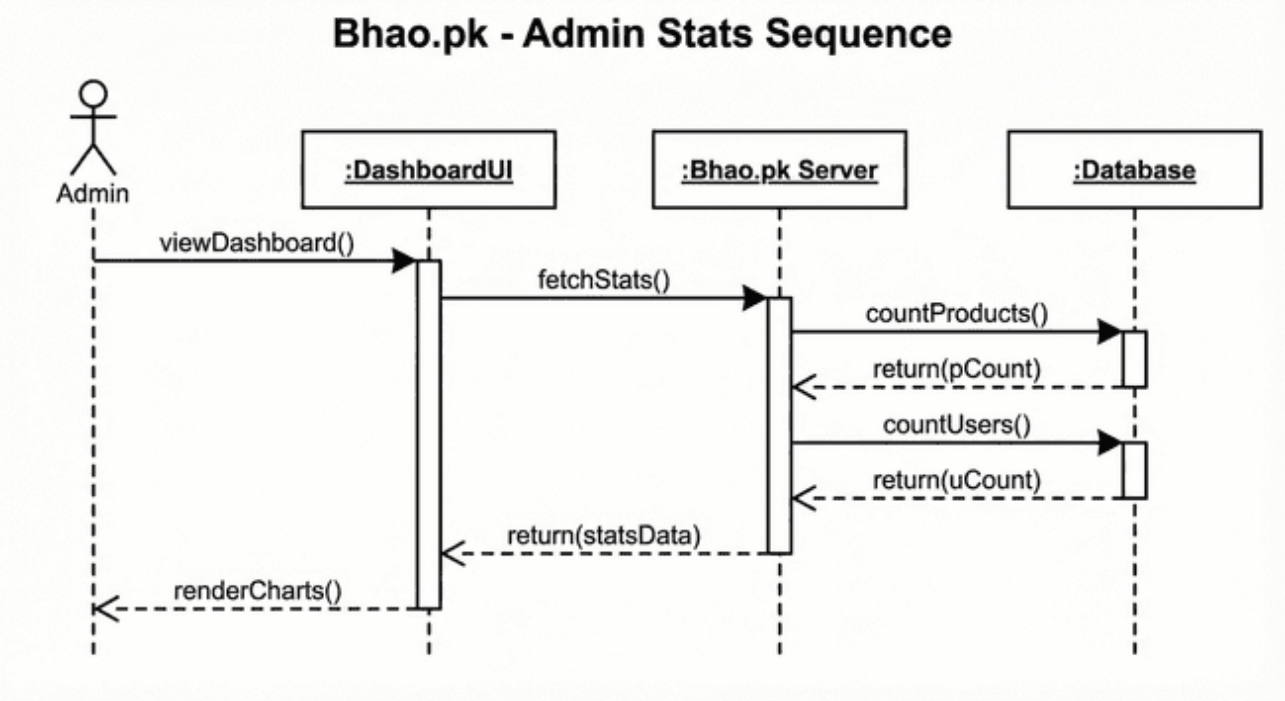
xxiii: Figure 3.8: Manage Wishlist Sequence Diagram



xxiv: Figure 3.9: Sign Up Sequence Diagram



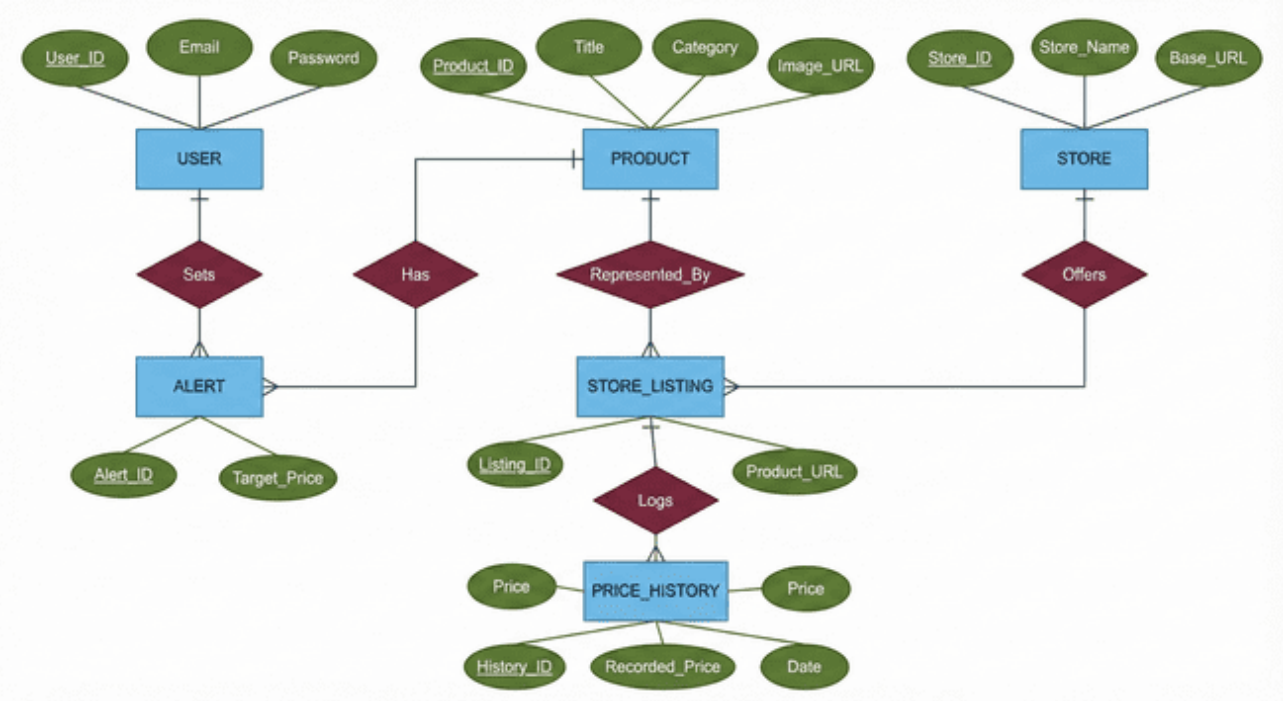
xxv: Figure 3.10: Login Sequence Diagram



xxvi: Figure 3.11: Sign Up Sequence Diagram

3.5. Entity Relationship Diagram

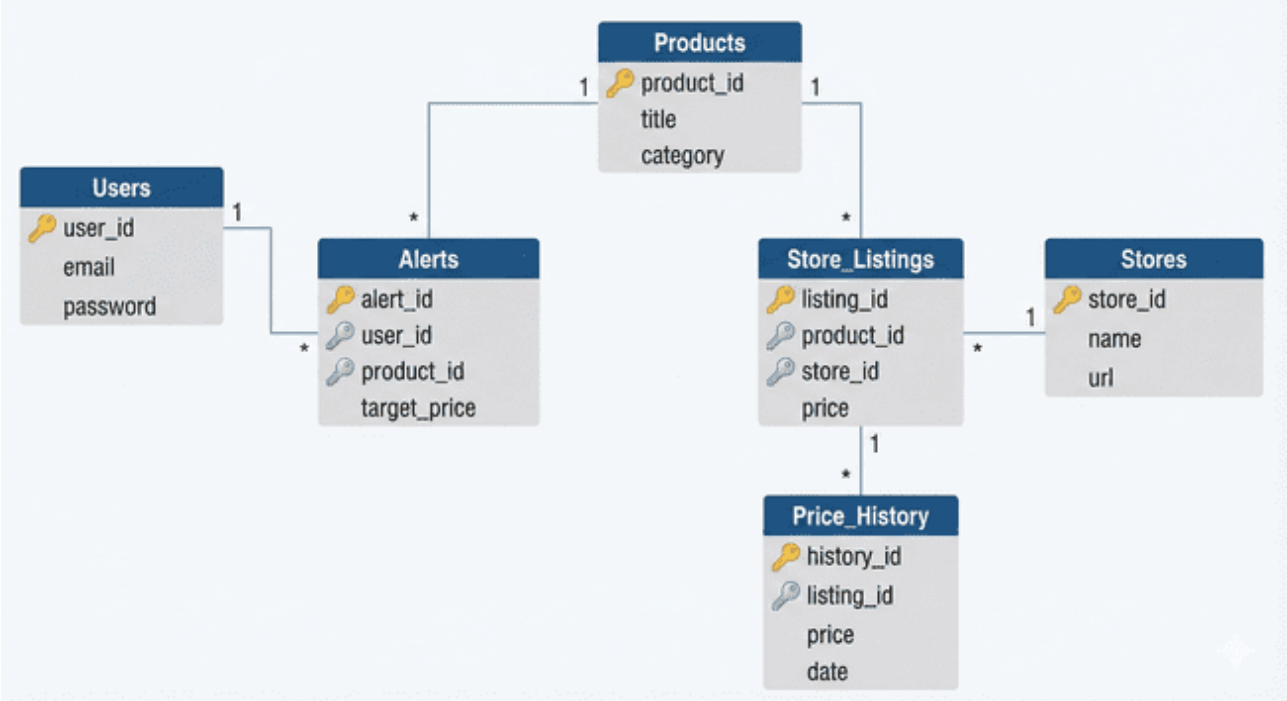
The Entity Relationship Diagram (ERD) for Bhao.pk visualizes the system's logical data structure, anchored by the **Product** entity which connects to multiple **StoreListings** to represent unique offers from various **Stores. Users** interact with the system by establishing **Alerts** on these products, creating a direct link between user preferences and market data. Additionally, the model incorporates a **PriceHistory** entity related to each listing, ensuring that every price fluctuation is recorded to support historical trend analysis.



xxvii: Figure 3.12: Entity Relations Diagram

3.6. Database Schema

The Database Schema for Bhao.pk represents the logical configuration of the relational database, structured around the central **Product** table which maintains a universal catalog of items. It utilizes a **Store_Listings** table to bridge these generic products with specific external **Stores**, allowing the system to manage multiple vendors and prices for a single item. **Users** are connected to specific products through an **Alerts** table to manage notifications, while a dedicated **Price_History** table archives temporal data for every listing to support historical trend analysis



xxviii: Figure 3.13: Entity Relations Diagram

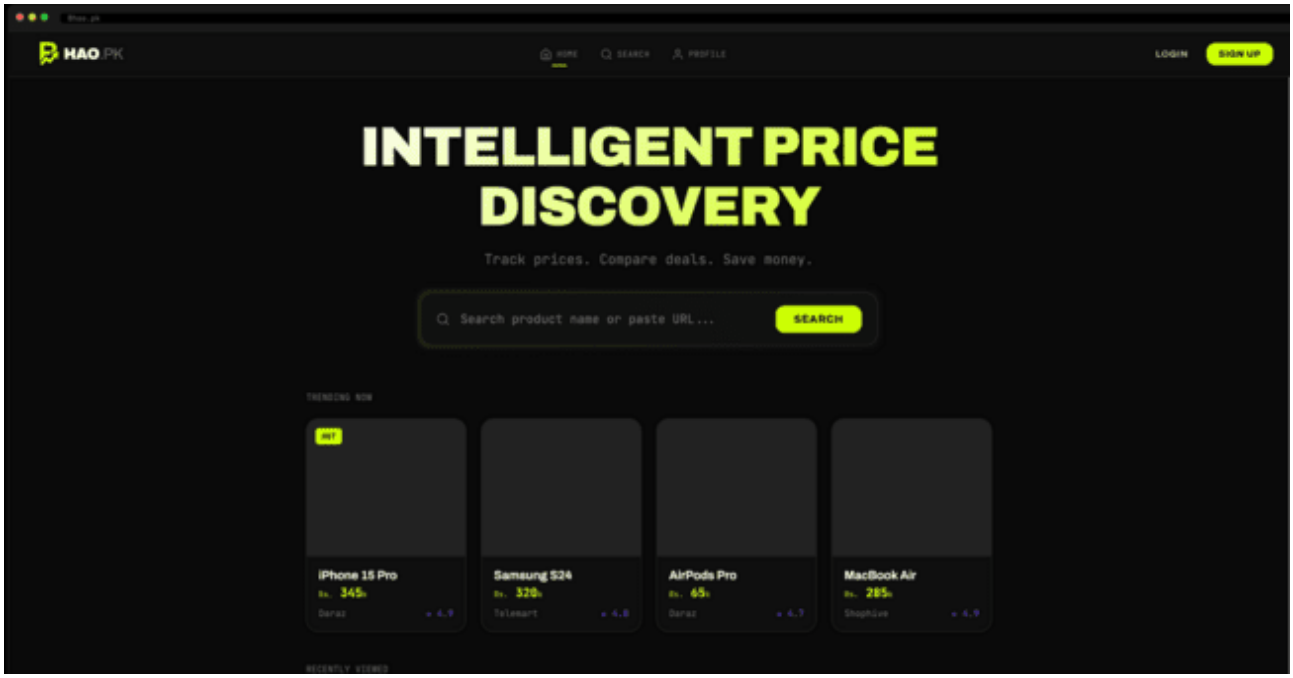
3.7. User Interface Design

The Bhao.pk user interface employs a sleek, dark-mode aesthetic with high-contrast neon accents to ensure readability and reduce eye strain. Its minimalist design prioritizes the search function and aggregates complex price data into clean, intuitive dashboards and graphs, enabling users to compare vendors and set alerts effortlessly.

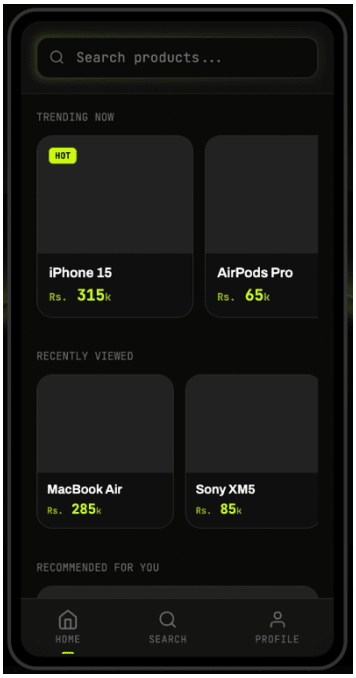
Homepage:

The Home Page serves as the primary entry point for Bhao.pk, featuring a minimalist, search-centric layout designed for immediate user engagement. Adopting a cohesive dark-mode aesthetic with neon accents, the web version centers around a prominent search bar and a "Trending Now" grid to encourage exploration. The mobile interface adapts this structure for smaller screens, placing the search bar at the top and utilizing a vertical scroll

layout, accompanied by a fixed bottom navigation bar for seamless accessibility across devices.



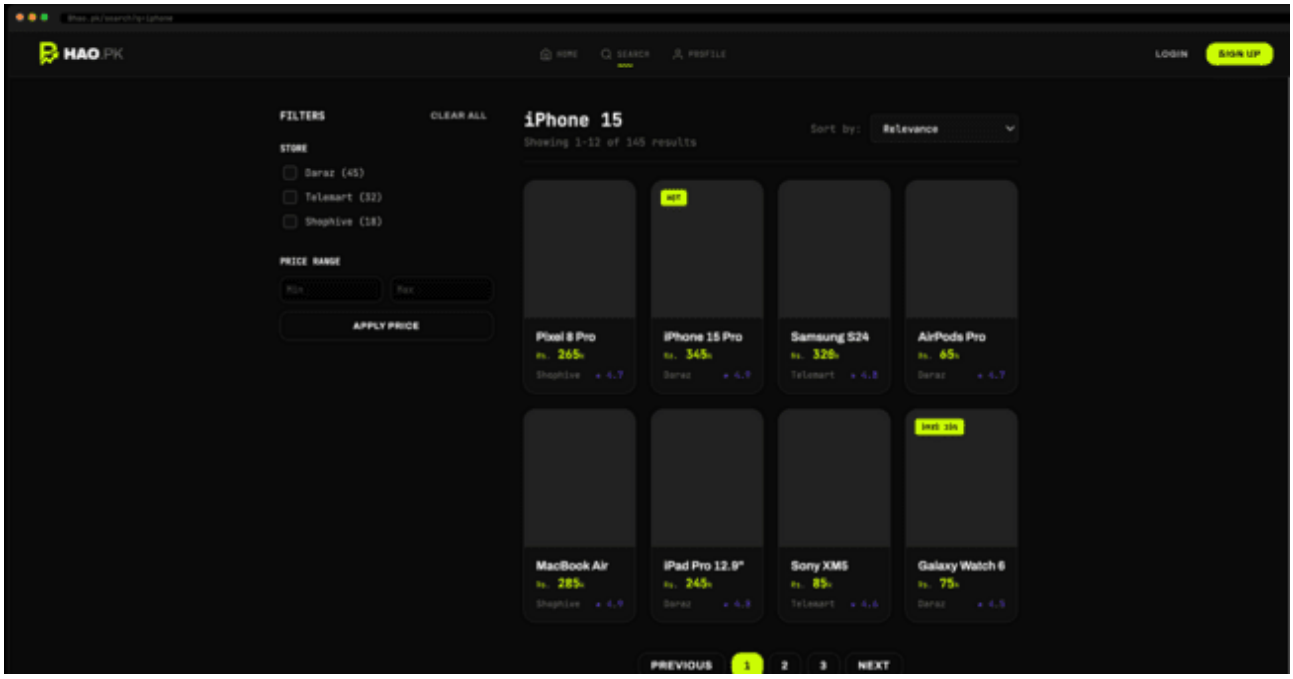
xxix: Figure 3.13: Web Homepage UI



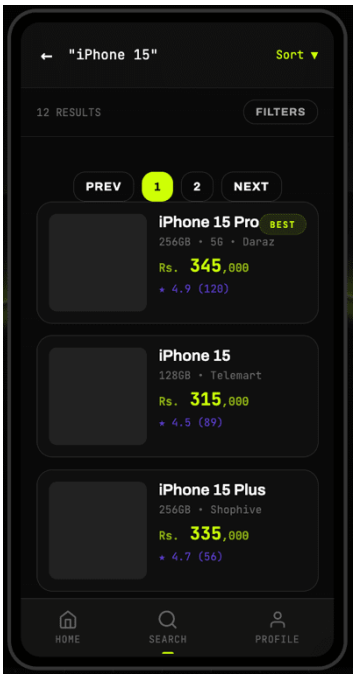
xxx: Figure3.14: Mobile Homepage UI

Search:

The Search Page provides a unified product discovery experience, tailored for both web and mobile platforms. On the web interface, the page features a robust sidebar with filters for price, store, and category, alongside a sorting dropdown, allowing users to efficiently refine their search results within a spacious grid layout. The mobile version optimizes this experience for smaller screens by condensing filters into a collapsible menu or modal, stacking product cards vertically for easy scrolling, and ensuring key actions like "Sort" and "Filter" are accessible via touch-friendly buttons at the top of the viewport.



xxxi: Figure 3.15: Mobile Search UI

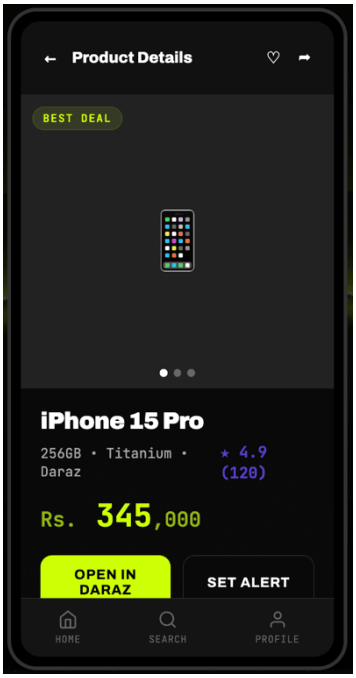


xxxii: Figure 3.16: Mobile Search UI

Product:

The Product Detail Page serves as the primary decision-making hub, presenting comprehensive product specifications and historical price trends in a unified, dark-mode interface. The web layout utilizes a split-screen design to showcase high-resolution imagery alongside an interactive price history graph and vendor comparison links. On mobile, this content is optimized into a vertical scroll, prioritizing price visibility and anchoring key actions like "Open in Store" and "Set Alert" at the bottom for seamless one-handed interaction.

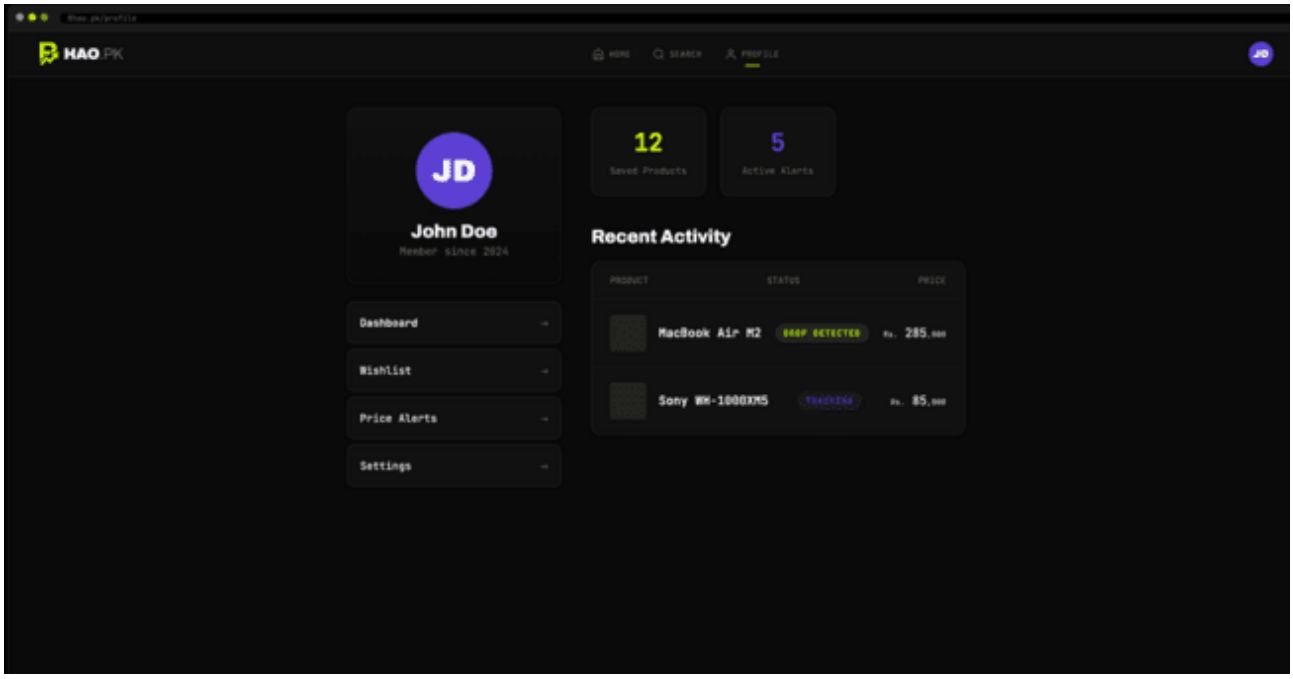
xxxiii: Figure 3.17: Web Product Page UIl



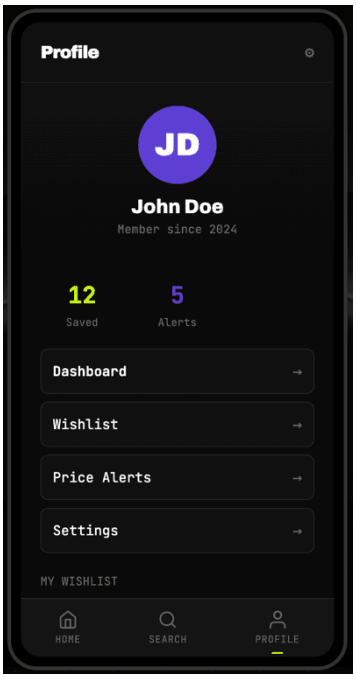
xxxiv: Figure 3.18: Mobile Product Page Ul

Profile:

The User Profile Page acts as a personalized command center, featuring a consistent dark-mode design across both platforms. On the web, it presents a dashboard-style layout with key metrics like "Active Alerts" and "Saved Items" displayed as prominent cards alongside a "Recent Activity" table. The mobile version streamlines this into a vertical menu with quick-access buttons for "Dashboard," "Wishlist," and "Price Alerts," ensuring users can manage their account settings and preferences efficiently on smaller screens.



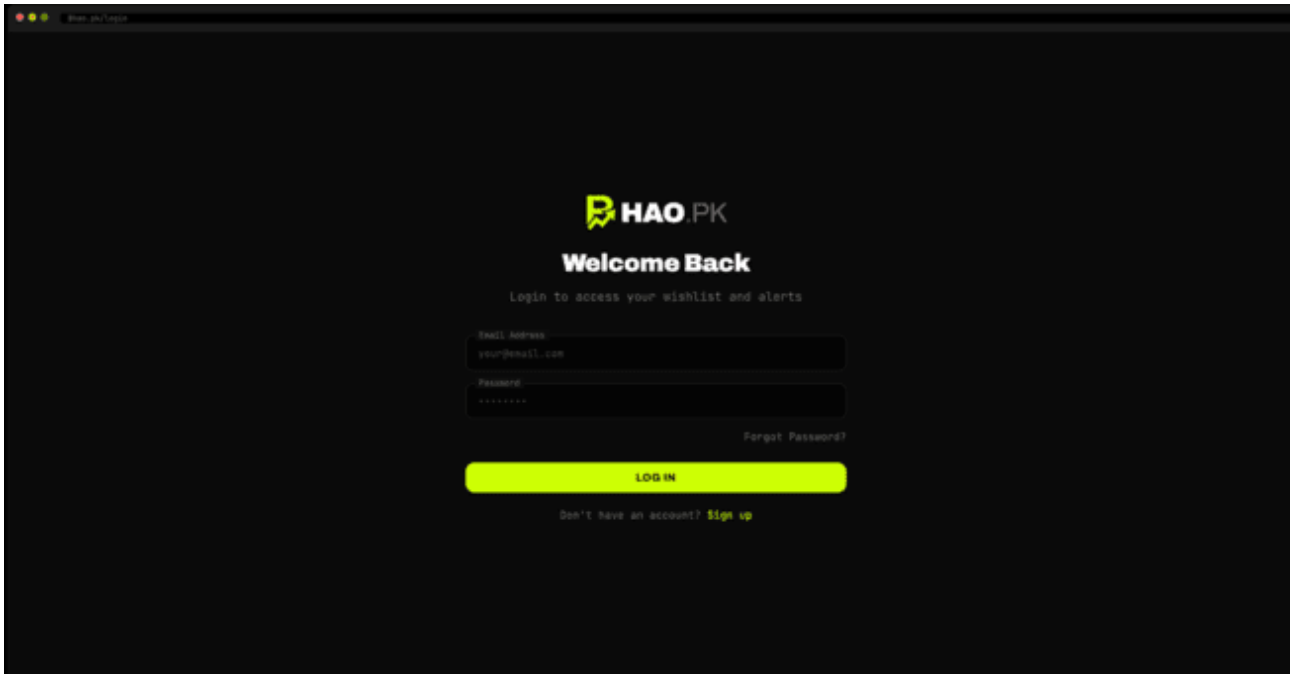
xxxv: Figure 3.19: Web Profile Page UI



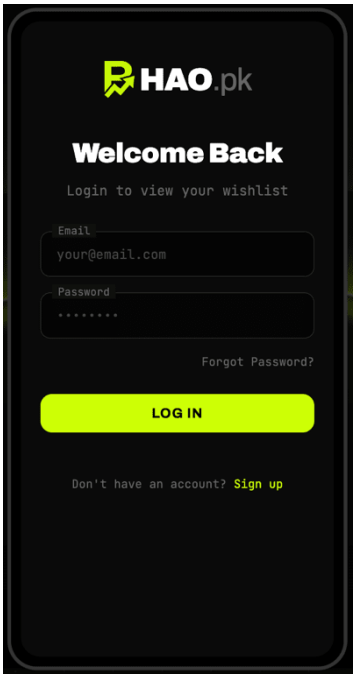
xxxvi: Figure 3.20: Mobile Profile Page UI

Login:

The Login Page provides a secure and intuitive entry point for registered users across both platforms. Adhering to the app's dark-mode aesthetic, it features a clean form for email and password input, prominent "Login" and "Forgot Password" actions, and clearly visible links for new user registration. On mobile, the layout is vertically centered with large, touch-friendly buttons, while the web version centers the form within a spacious, branded container to maintain focus.



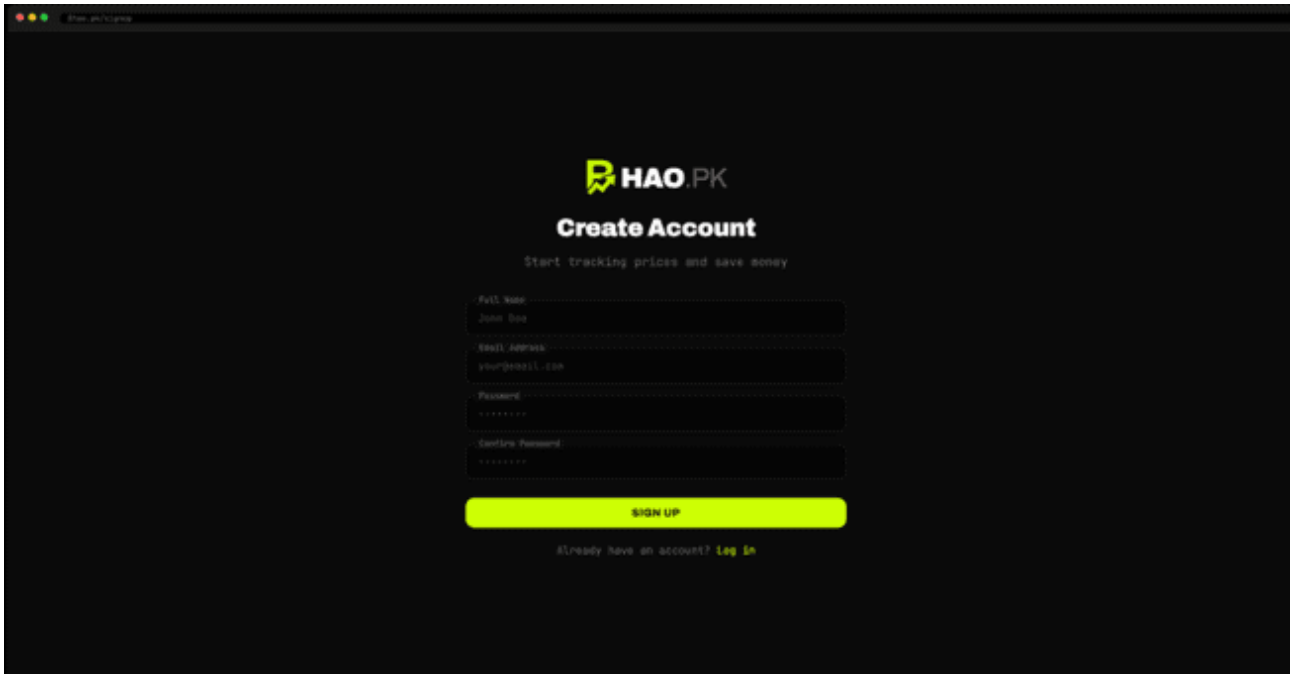
xxxvii: Figure 3.21: Web Login Page UI



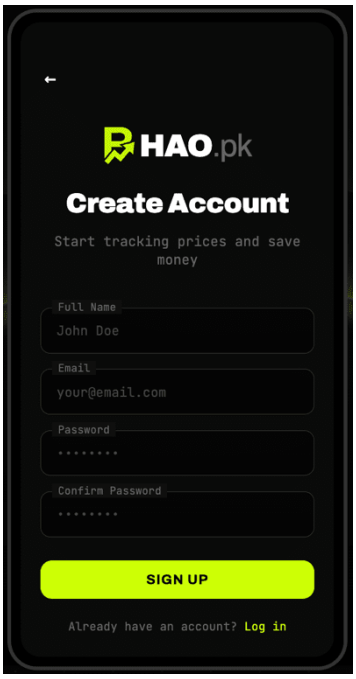
xxxviii: Figure 3.22: Mobile Login Page UI

Sign Up:

The Signup Page offers a streamlined registration process for new users, maintaining a consistent dark-mode aesthetic across platforms. Both the web and mobile interfaces feature a simple form for "Full Name," "Email," and "Password" (with confirmation), clearly labeled input fields, and a prominent neon "Sign Up" button. The design minimizes friction by including clear error validation and a direct link to the "Login" page for existing users, ensuring a smooth onboarding experience.



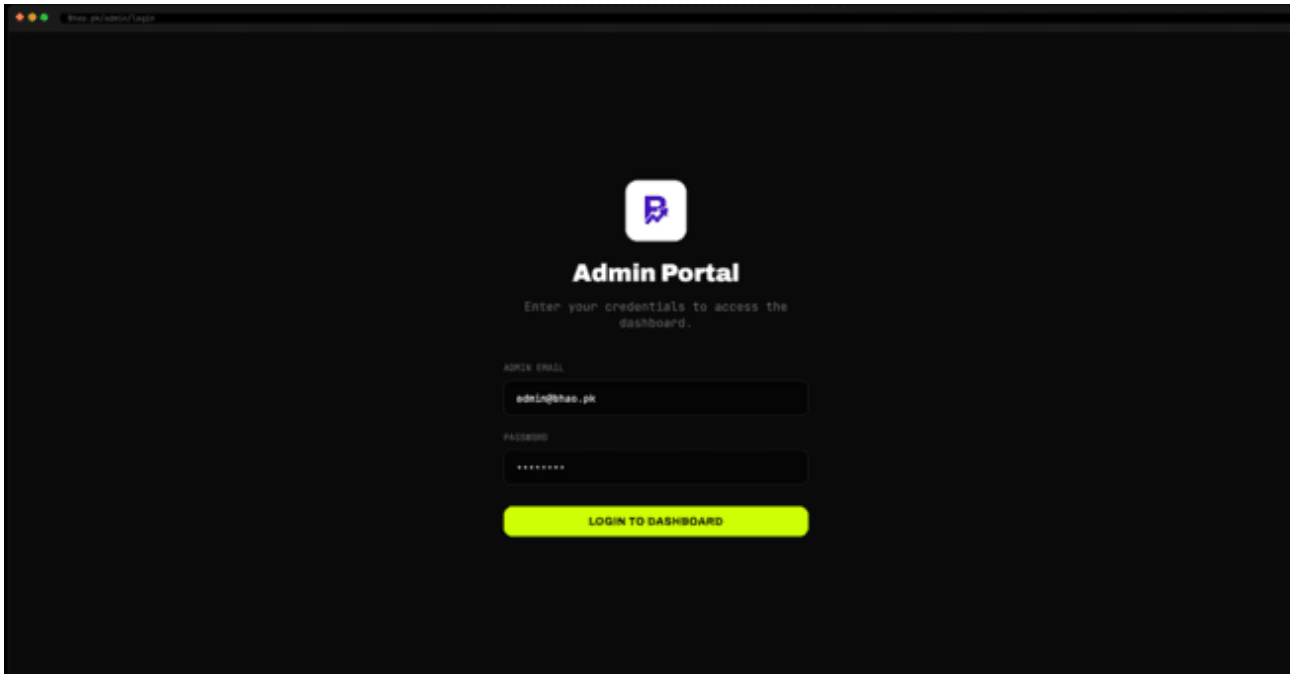
xxxix: Figure 3.23: Web Sign Up Page UI



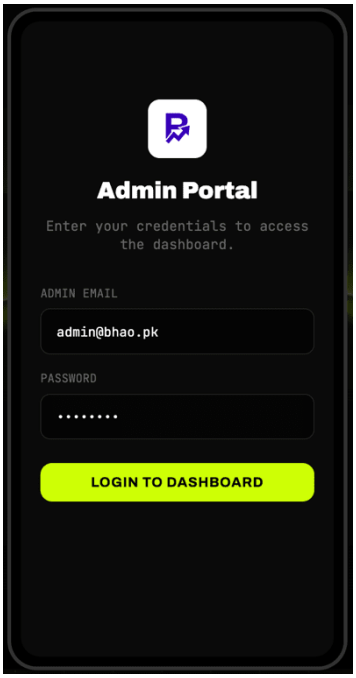
xl: Figure 3.24: Mobile Sign Up Page UI

Admin Login:

The Admin Login Page is a secure, restricted-access portal designed with the same cohesive dark-mode aesthetic as the user-facing application. It features a simplified login form requiring specific admin credentials, stripped of social login options or registration links to ensure security. The interface is optimized for both desktop and mobile, providing administrators with a direct gateway to the backend dashboard for monitoring system health and statistics.



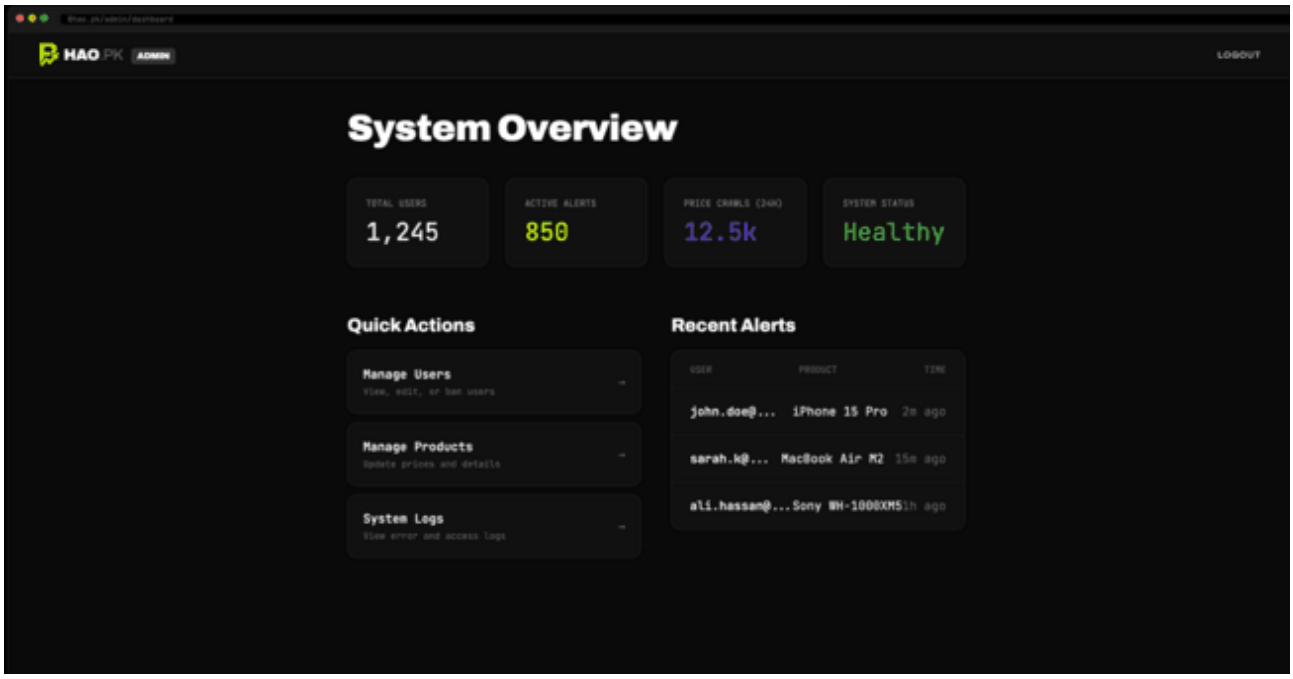
xli: Figure 3.25: Web Admin Login Page UI



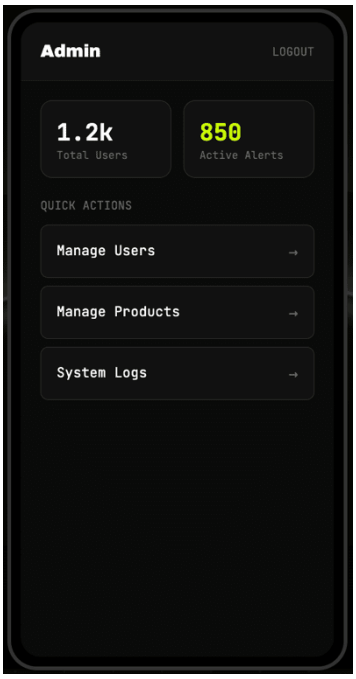
xlii: Figure 3.26: Mobile Admin Login Page UI

Admin Dashboard:

The Admin Dashboard Page acts as a centralized command center, accessible securely on both web and mobile platforms. The interface provides real-time oversight of system health, featuring prominent metrics for total active users, scraped products, and system uptime. The design is optimized for rapid assessment, with interactive charts visualizing user growth and scraper performance, alongside quick-action tools for managing user accounts and system configurations.



xliii: Figure 3.27: Web Admin Login Page UI



xliv: Figure 3.28: Mobile Admin Login Page UI

Chapter 4

Software Development

This chapter documents the coding standards, development environment, and software description for the Bhao.pk platform. It covers the implementation details for 30% of the functional requirements (8 out of 27), including the architecture of each module, relevant code excerpts from the actual codebase, and explanations of how each feature operates.

4.1. Coding Standards

The Bhao.pk codebase follows a consistent set of coding standards across both the mobile application (React Native with Expo) and the web application (Next.js). All source code is written in TypeScript, which provides compile-time type checking and improved developer experience. The following subsections describe the adopted conventions.

4.1.1. Indentation

Two spaces are used as the standard unit of indentation throughout the project, consistent with the JavaScript/TypeScript community convention. All nested blocks, including function bodies, conditional statements, JSX elements, and object literals, follow this indentation pattern consistently. The project's TypeScript configuration (tsconfig.json) enforces strict mode to maintain code quality.

xlv: Listing 4.1: Indentation Standard

```
// Consistent 2-space indentation example
const handleLogin = async () => {
  const formData = { email, password };
  const validationErrors = validateForm(formData, loginSchema);
  if (Object.keys(validationErrors).length > 0) {
    setErrors(validationErrors);
    return;
  }
  setLoading(true);
  try {
    await login(email, password);
    navigation.replace("Main");
  } catch (error) {
```

```
    console.error("Login error:", error);
  }
};
```

4.1.2. Declaration

Variable declarations follow the modern ECMAScript convention. The const keyword is used for all variables that are not reassigned, and let is used for mutable variables. The var keyword is never used. React state variables are declared using the useState hook with explicit TypeScript type annotations. Interfaces and type aliases are declared in dedicated type files (src/types/) and imported where needed.

xlvi: Listing 4.2: Declaration Standards

```
// Type-safe declarations
const [user, setUser] = useState<User | null>(null);
const [isLoading, setIsLoading] = useState(true);
const [errors, setErrors] = useState<Record<string, string>>({});

// Interface declaration (src/types/models.ts)
interface User {
  id: string;
  name: string;
  email: string;
  createdAt: string;
  profilePicture?: string;
}
```

4.1.3. Statement Standards

Each line contains at most one statement. Arrow functions are preferred over traditional function declarations for callbacks and inline functions. Async/await syntax is used for all asynchronous operations instead of raw Promises or callbacks. Error handling is implemented using try-catch blocks around all async operations, with user-facing error messages displayed via toast notifications.

xlvii: Listing 4.3: Statement Standards with Async/Await

```
// Async/await with error handling
const login = async (email: string, password: string) => {
  try {
    const response = await authService.login(email, password);
    await AsyncStorage.setItem("@auth_user", JSON.stringify(response.user));
    await AsyncStorage.setItem("@auth_token", response.token);
    apiClient.setAuthToken(response.token);
```

```
    setUser(response.user);
    Toast.show({ type: "success", text1: "Welcome back!" });
  } catch (error) {
    Toast.show({ type: "error", text1: "Login failed" });
    throw error;
  }
};
```

4.1.4. Naming Conventions

The project adheres to the following naming conventions consistently:

Components and Screens: PascalCase (e.g., LoginScreen, ProductCard, SearchScreen)

Hooks: camelCase with 'use' prefix (e.g., useSearch, useWishlist, useAuth)

Services: camelCase with descriptive suffix (e.g., authService, alertService, apiClient)

Constants: UPPER_SNAKE_CASE (e.g., COLORS, SPACING, BORDER_RADIUS, ALL_PRODUCTS)

Type Interfaces: PascalCase (e.g., User, ProductWithListings, SearchFilters)

File names: PascalCase for components (LoginScreen.tsx), camelCase for utilities (validation.ts)

State variables: camelCase with descriptive names (e.g., isAuthenticated, filteredProducts)

4.2. Development Environment

The Bhao.pk platform is developed as a cross-platform system consisting of a React Native mobile application and a Next.js web application. The following tools and technologies constitute the development environment:

Tool / Technology	Purpose & Version
Visual Studio Code	Primary IDE for TypeScript, React Native, and Next.js development
React Native 0.81.5	Cross-platform mobile framework for Android and iOS
Expo SDK 54	Development platform providing build tools, native APIs, and testing utilities

Next.js (React 19.1.0)	Server-side rendered web framework for the web dashboard
TypeScript 5.9.2	Statically typed superset of JavaScript for type safety
Node.js	JavaScript runtime for development server and build processes
EAS CLI	Expo Application Services for building production APK/AAB files
Android Emulator	Virtual device for testing the mobile application
Google Chrome DevTools	Browser-based debugging and testing for the web application
Git / GitHub	Version control and collaborative development
AsyncStorage	On-device key-value storage for authentication tokens and user data

xvii: Table 4.1: Development Environment

Visual Studio Code was chosen as the primary IDE due to its excellent TypeScript integration, built-in terminal, rich extension ecosystem (ESLint, Prettier, React Native Tools), and lightweight performance. The Expo framework was selected to accelerate React Native development by providing pre-configured build pipelines, over-the-air updates, and access to native device APIs without manual native code configuration.

The dual-platform architecture (React Native for mobile, Next.js for web) was adopted to maximize code reuse. Both applications share common TypeScript type definitions (src/types/), service layers (src/services/), and custom hooks (src/hooks/), ensuring consistent business logic across platforms.

4.3. Database Management System

The Bhao.pk system employs a dual-storage strategy comprising a client-side local storage layer and a backend API architecture designed for future database integration.

On the client side, React Native AsyncStorage serves as the primary local data persistence mechanism. It stores authentication tokens, user session data, wishlist items, and application preferences as key-value pairs. AsyncStorage provides asynchronous, unencrypted, persistent storage that persists across application restarts.

xlviii: Listing 4.5: AsyncStorage Data Persistence

```
// Storage abstraction for authentication and wishlist
await AsyncStorage.setItem("@auth_user", JSON.stringify(response.user));
await AsyncStorage.setItem("@auth_token", response.token);

const wishlistStorage = {
  getWishlist: async () => {
    const data = await AsyncStorage.getItem("@wishlist");
    return data ? JSON.parse(data) : [];
  },
};
```

The API client (src/services/api/client.ts) implements a dual-mode architecture that supports both mock data for development and real HTTP requests for production. In mock mode, the client intercepts all API calls and returns predefined data from the mockDataService.

xlix: Listing 4.6: API Client with Mock/Production Modes

```
// src/services/api/client.ts - Dual-mode API client
class ApiClient {
  private baseUrl: string;
  private token: string | null = null;

  private async request<T>(endpoint: string, options?: RequestInit): Promise<T> {
    if (USE_MOCK_DATA) {
      return mockDataService.getData<T>(endpoint, options);
    }
    const headers: HeadersInit = {
      "Content-Type": "application/json",
      ...(this.token && { Authorization: `Bearer ${this.token}` }),
    };
    const response = await fetch(`${this.baseUrl}${endpoint}`, { ...options, headers });
    return response.json();
  }
}
```

4.4. Software Description

This section describes the implementation of 8 functional requirements (30% of the total 27), covering the core modules of the Bhao.pk platform. Each module is presented with its

purpose, input/output specification, relevant code excerpts, and an explanation of the implementation logic.

4.4.1. Product Search Module (FR-6)

The Product Search module enables users to search for products by entering keywords in the search bar. The search engine performs real-time matching against product names, specifications, and categories across all supported Pakistani e-commerce stores (Daraz, Telemart, Shophive).

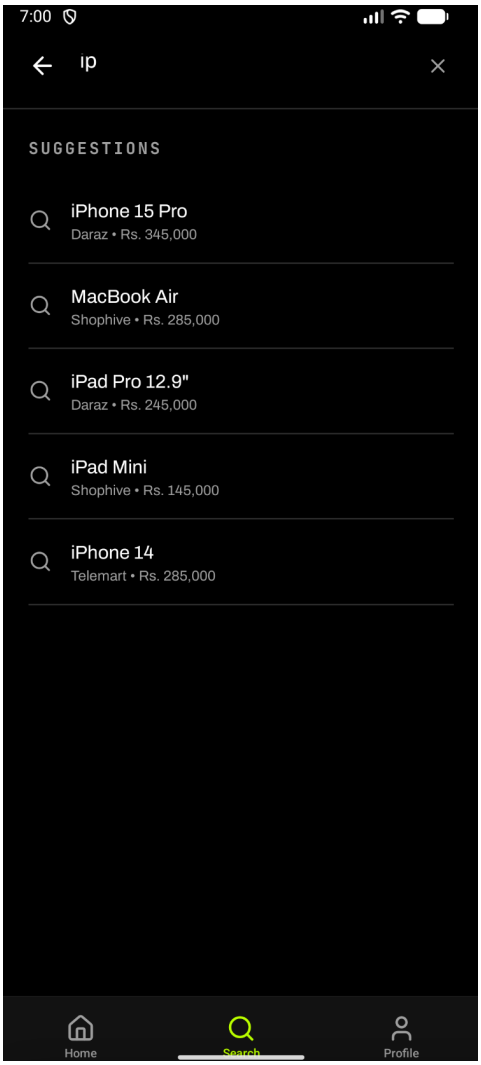
Input:
The user enters a text query (e.g., 'USB-C Charger') in the search bar on the SearchScreen.

Output:
A filtered list of ProductCard components is displayed, each showing the product name, image, starting price, store badge, and rating.

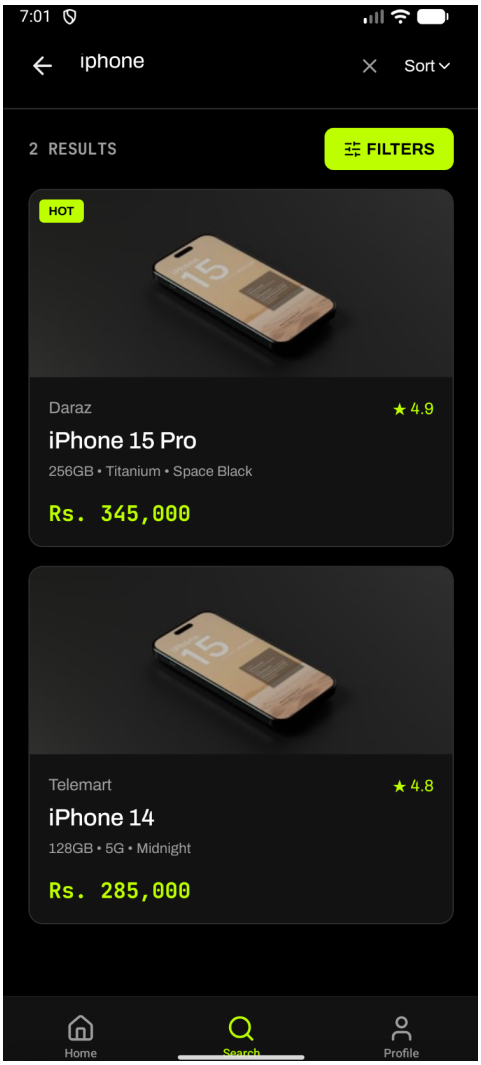
I: Listing 4.7: Product Search Implementation

```
// src/hooks/useSearch.ts - Core filtering logic
const filterAndSort = () => {
  let filtered = [...allProducts];
  if (query.trim()) {
    const lowerQuery = query.toLowerCase();
    filtered = filtered.filter(
      (product) =>
        product.name.toLowerCase().includes(lowerQuery) ||
        product.specs?.toLowerCase().includes(lowerQuery) ||
        product.category?.toLowerCase().includes(lowerQuery)
    );
  }
  setResults(filtered);
};
```

The search logic is separated into a reusable useSearch custom hook that accepts the full product array and returns filtered results. This separation of concerns allows the same search logic to be used across both the mobile and web applications.



Ii: Figure 4.1: Search Screen with Real-time Suggestions



lii: Figure 4.2: Search Results for 'USB-C Charger'

4.4.2. Price Sorting Module (FR-8)

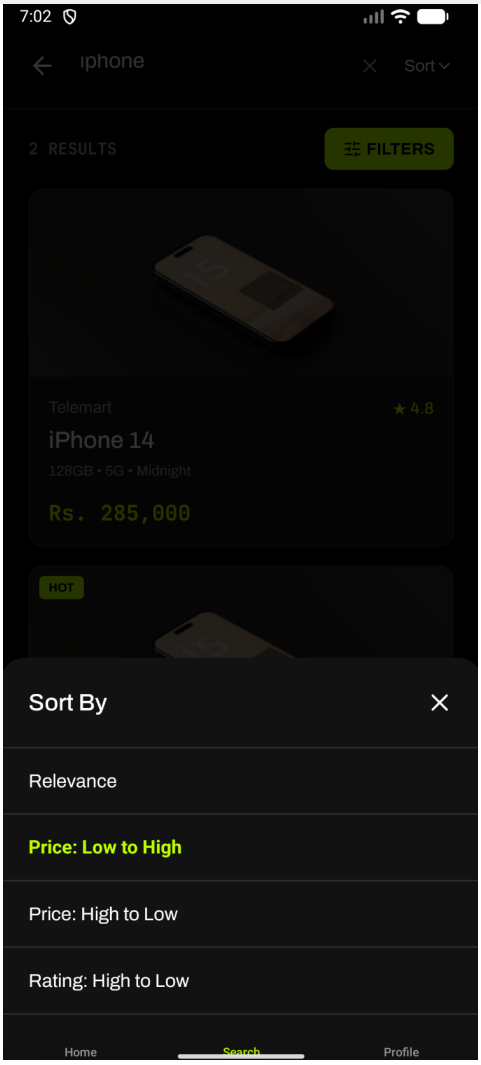
The Price Sorting module allows users to reorder search results by price. Options include Relevance, Price: Low to High, Price: High to Low, and Rating.

Input:
The user taps the 'Sort' button and selects 'Price: Low to High' from the modal.

Output:
The product list is reordered with the lowest-priced product appearing first.

liii: Listing 4.8: Price Sorting Logic

```
// src/hooks/useSearch.ts - Sorting implementation
case "price_asc":
  filtered.sort((a, b) => {
    const priceA = parseFloat(a.price?.replace(/[^0-9]/g, "") || "0");
    const priceB = parseFloat(b.price?.replace(/[^0-9]/g, "") || "0");
    return priceA - priceB;
  });
  break;
```



liv: Figure 4.3: Sort Options Modal

4.4.3. Store Filter Module (FR-9)

The Store Filter module enables users to narrow search results to products from specific e-commerce platforms.

Input:

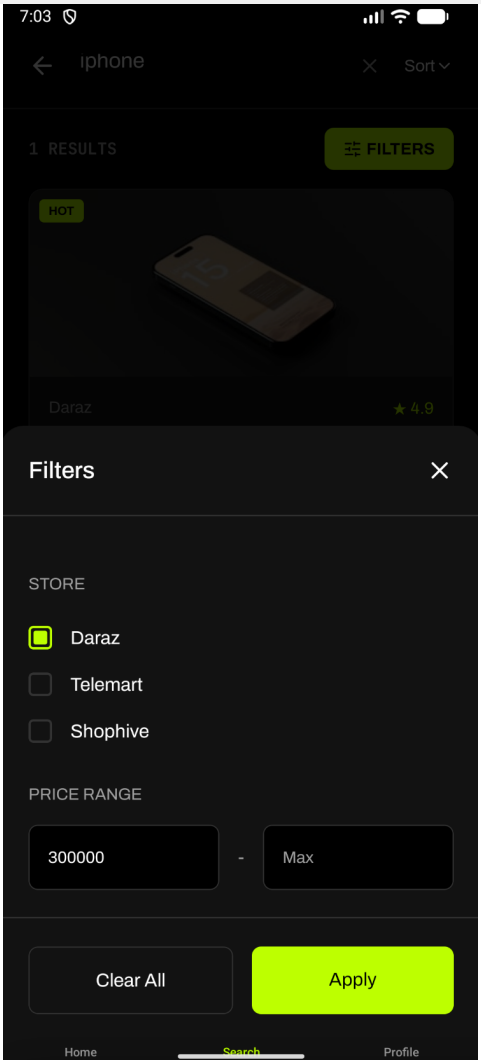
The user opens the Filters modal and selects one or more stores (Daraz, Telemart, Shophive).

Output:

Only products available on the selected store(s) are displayed.

lv: Listing 4.9: Store Filter Implementation

```
// src/hooks/useSearch.ts - Store filtering
if (filters.stores && filters.stores.length > 0) {
  filtered = filtered.filter((product) =>
    filters.stores!.includes(product.store || "")
  );
}
```



lvi: Figure 4.4: Filter Modal with Store Selection

4.4.4. User Registration Module (FR-13)

The User Registration module allows new users to create an account by providing their name, email address, and password with client-side form validation.

Input:

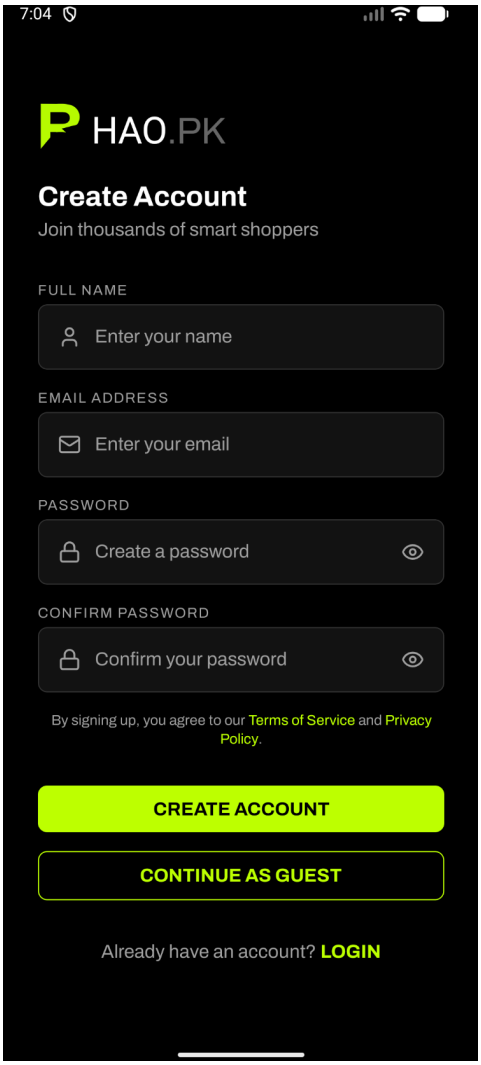
The user enters their full name, email, password, and password confirmation on the SignupScreen.

Output:

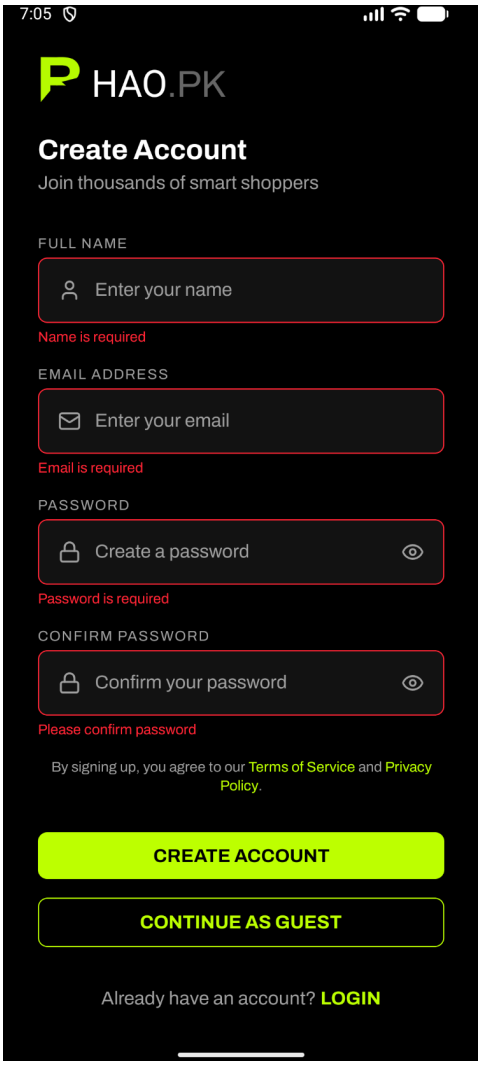
Upon successful validation, the user is authenticated and redirected to the main application.

lvii: Listing 4.10: Signup Validation Schema

```
// src/utils/validation.ts - Signup validation schema
export const signupSchema: ValidationSchema = {
  name: [
    validators.required("Name is required"),
    validators.minLength(2, "Name must be at least 2 characters"),
  ],
  email: [validators.required("Email is required"), validators.email()],
  password: [validators.required("Password is required"), validators.password()],
  confirmPassword: [
    validators.required("Please confirm password"),
    validators.match("password", "Passwords do not match"),
  ],
};
```



lviii: Figure 4.5: Sign Up Screen



lix: Figure 4.6: Sign Up with Validation Errors

4.4.5. User Authentication Module (FR-14)

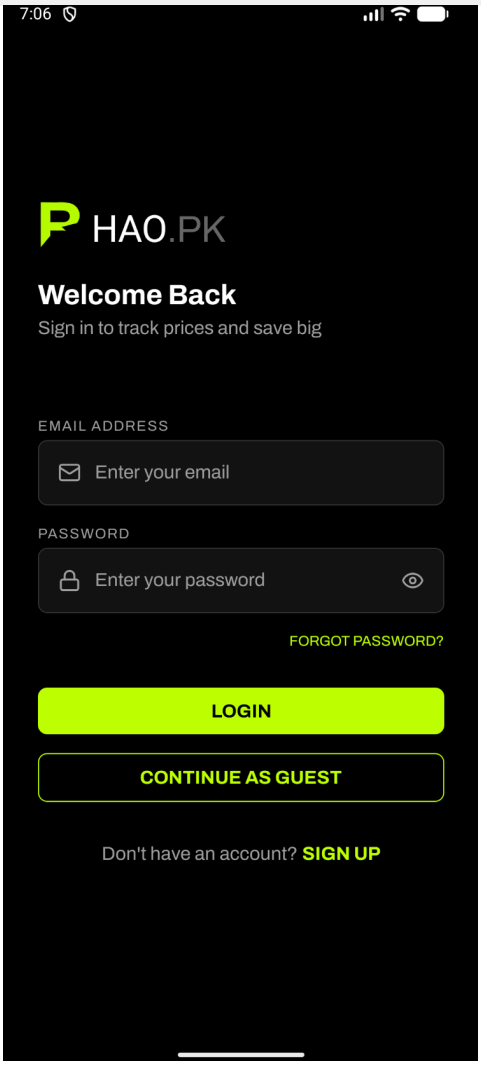
The User Authentication module handles user login using email and password credentials, integrating with the AuthContext provider for global authentication state.

Input:
The user enters their email and password on the LoginScreen.

Output:
The JWT token is persisted, auth state is updated, and the user is navigated to the home screen.

lx: Listing 4.11: Login Implementation

```
// src/context/AuthContext.tsx - Login handler
const login = async (email: string, password: string) => {
  try {
    const response = await authService.login(email, password);
    await AsyncStorage.setItem("@auth_user", JSON.stringify(response.user));
    await AsyncStorage.setItem("@auth_token", response.token);
    apiClient.setAuthToken(response.token);
    setUser(response.user);
    Toast.show({ type: "success", text1: "Welcome back!" });
  } catch (error) {
    Toast.show({ type: "error", text1: "Login failed" });
    throw error;
  }
};
```



lxi: Figure 4.7: Login Screen

4.4.6. User Logout Module (FR-15)

The Logout module allows authenticated users to terminate their session.

Input:

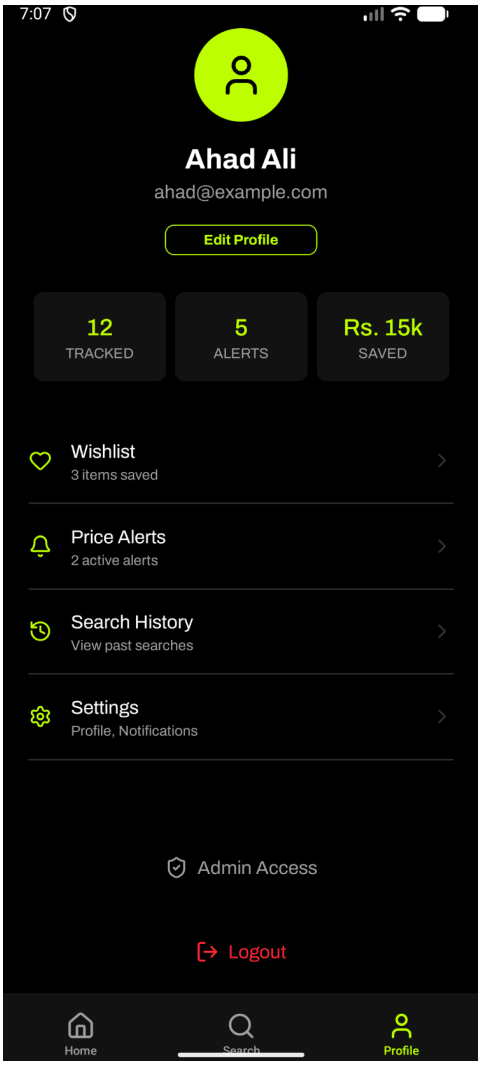
The user navigates to Profile and taps 'Logout'.

Output:

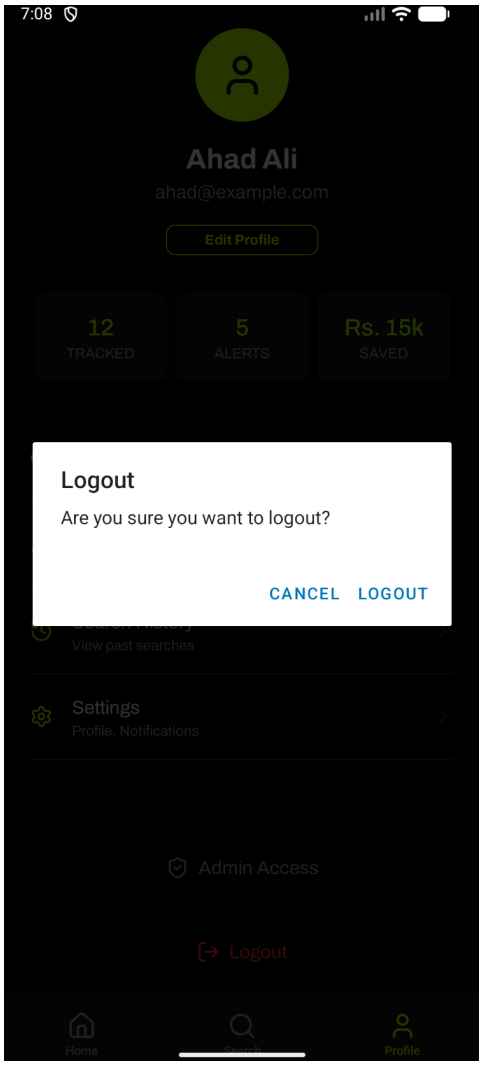
A confirmation dialog appears. Upon confirming, session is cleared and user returns to Login screen.

lxii: Listing 4.12: Logout Implementation

```
// src/screens/ProfileScreen.tsx - Logout with confirmation
const handleLogout = () => {
  Alert.alert(
    "Logout",
    "Are you sure you want to logout?",
    [
      { text: "Cancel", style: "cancel" },
      { text: "Logout", style: "destructive",
        onPress: () => navigation.replace("Login") }
    ]
  );
};
```



lxiii: Figure 4.8: Profile Screen with Logout Button



lxiv: Figure 4.9: Logout Confirmation Dialog

4.4.7. Wishlist Management Module (FR-21)

The Wishlist module allows users to save products for later reference via the heart icon on product cards.

Input:

The user taps the heart icon on a product card.

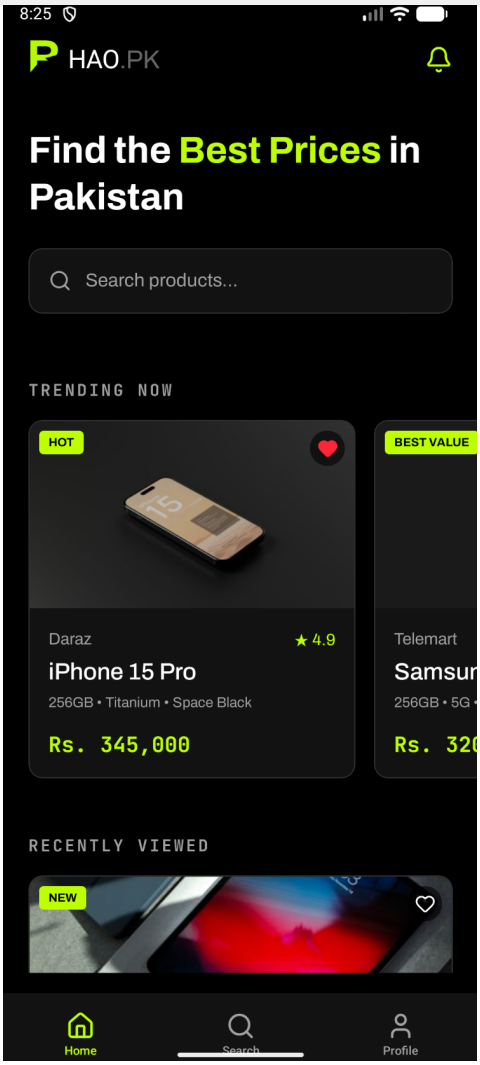
Output:

The product is added/removed from the wishlist with a toast notification. Data persists via AsyncStorage.

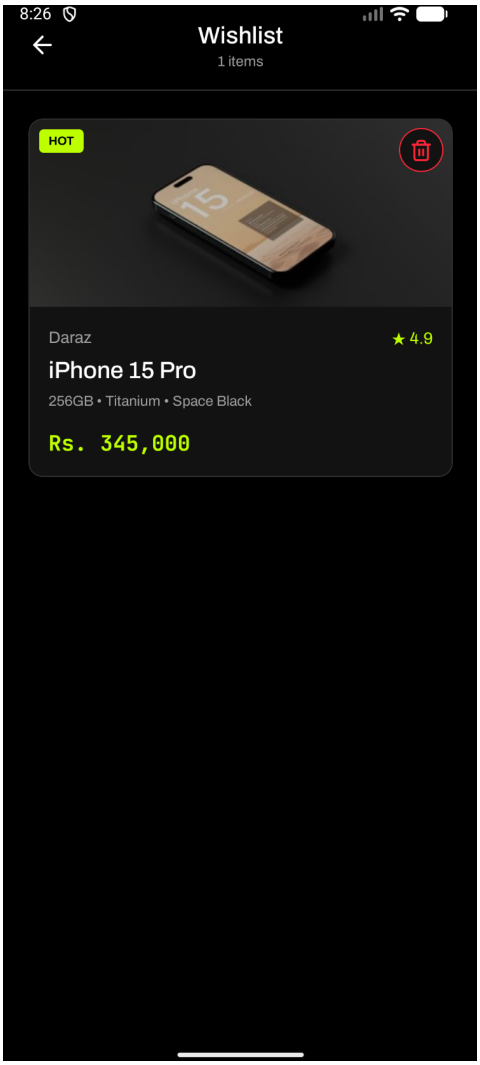
lxv: Listing 4.13: Wishlist Hook Implementation

```
// src/hooks/useWishlist.ts - Wishlist management
const addToWishlist = async (productId: string) => {
  await wishlistStorage.addToWishlist(productId);
  setWishlist((prev) => [...prev, productId]);
  Toast.show({ type: "success", text1: "Added to wishlist" });
};

const toggleWishlist = async (productId: string) => {
  if (wishlist.includes(productId)) {
    await removeFromWishlist(productId);
  } else {
    await addToWishlist(productId);
  }
};
```



lxvi: Figure 4.10: Product Card with Wishlist Heart Icon



lxvii: Figure 4.11: Wishlist Screen

4.4.8. Price Alert Module (FR-17)

The Price Alert module enables users to set a target price for any product and receive notifications when the price drops.

Input:

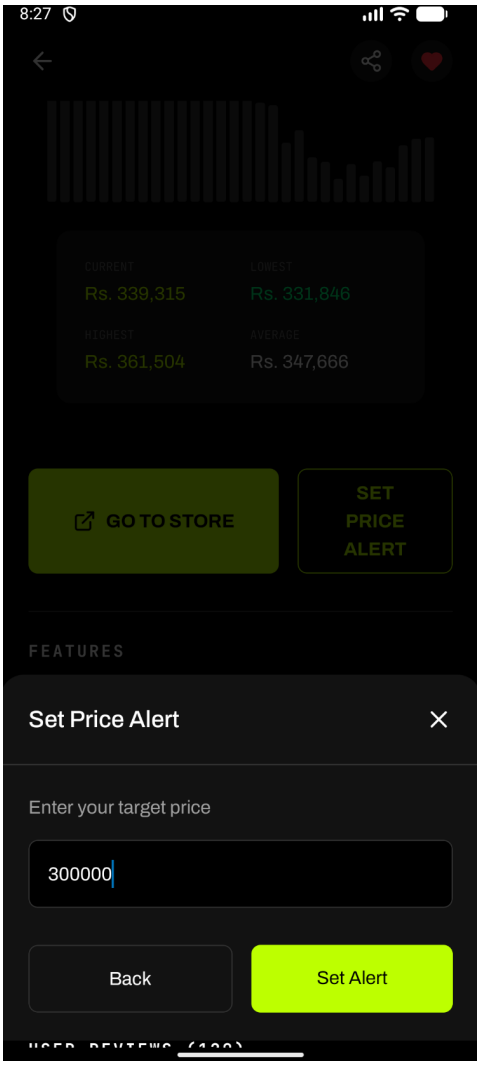
The user enters a target price on the Product Detail screen and taps 'Set Alert'.

Output:

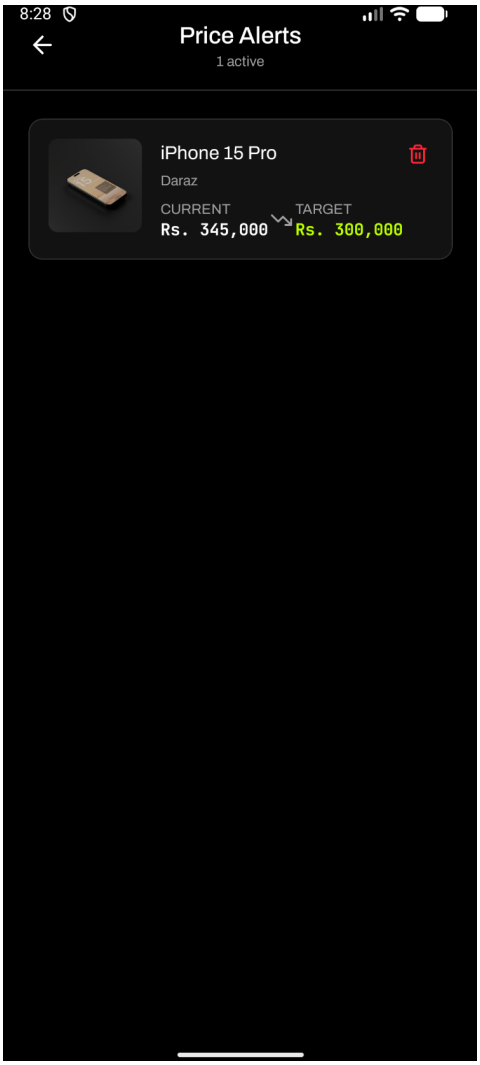
An alert record is created and appears in the Alerts screen with product name, current price, and target price.

lxviii: Listing 4.14: Alert Service Implementation

```
// src/services/api/alert.service.ts - Alert CRUD
export const alertService = {
  async createAlert(
    productId: string,
    targetPrice: number
  ): Promise<CreateAlertResponse> {
    return await apiClient.post<CreateAlertResponse>("/alerts", {
      productId, targetPrice,
    });
  },
  async deleteAlert(alertId: string): Promise<void> {
    await apiClient.delete(`/alerts/${alertId}`);
  },
  async toggleAlert(alertId: string, isActive: boolean): Promise<Alert> {
    return await apiClient.put<Alert>(`/alerts/${alertId}`, { isActive });
  },
};
```



lxix: Figure 4.12: Product Detail - Set Alert



lxx: Figure 4.13: Alerts Management Screen

Chapter 5

Software Testing

This chapter provides a comprehensive description of the testing procedures adopted during the development of Bhao.pk. It includes the selected testing methodology, the testing environment, and detailed test cases with results for the functional requirements addressed in the current iteration.

5.1. Testing Methodology

The testing phase of Bhao.pk employs Black Box Testing as the primary methodology. Black box testing examines the functionality of an application without examining its internal code structure. This method focuses solely on the inputs and outputs of the system, verifying that the software behaves according to its specified requirements.

Black box testing was selected for the following reasons:

It validates the system from the end-user's perspective, ensuring all features function correctly.

Test cases are designed based on functional requirements, making them directly traceable to specifications.

The tester does not require knowledge of the underlying React Native or TypeScript implementation.

It is effective for testing user interface interactions, form validations, and navigation flows.

It facilitates early detection of missing or incorrect functionality before deployment.

The testing scope covers 30% of the total functional requirements (8 out of 27), focusing on core features including user authentication, product search and filtering, wishlist management, and price alert creation.

5.2. Testing Environment

The Bhao.pk system consists of two client applications: a React Native mobile app built with Expo SDK 54 and a Next.js web application. Testing was conducted on the following environments:

Component	Details
Mobile Framework	React Native 0.81.5 with Expo SDK 54
Web Framework	Next.js (React 19.1.0)
Programming Language	TypeScript 5.9.2
Mobile Testing Device	Android Emulator / Physical Device
Web Testing Browser	Google Chrome (Latest)
Operating System	macOS / Windows 11
API Mode	Mock Data Service (Development Mode)

xviii: Table 5.1: Testing Environment Specification

The application operates in mock data mode during testing, where the API client returns predefined data from the mockDataService. This ensures consistent and reproducible test results.

5.3. Test Cases

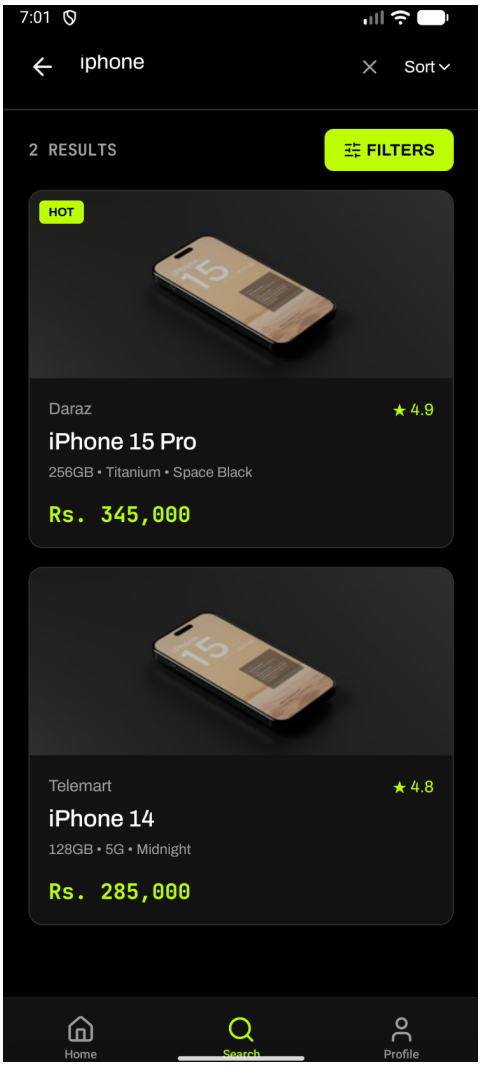
The test cases below correspond to 30% of the functional requirements from Chapter 2. Each test case verifies a specific user-facing feature of the Bhao.pk platform.

5.3.1. Test Case: Search for Products Using Keywords

xix: Table 5.2: Search for Products Using Keywords

Date:	10 February 2026
System:	Bhao.pk
Objective:	Verify that the user can search for products by entering keywords
Test ID:	1
Version:	1.0
Test Type:	Black Box - Functional
Input:	Search query: "Iphone"
Expected Result:	The system displays matching products from multiple stores with name, image, price, and store badge.
Actual Result:	**Passed**

The search functionality uses the useSearch hook for managing query state and filtering logic. When the user types a keyword, the hook filters the product dataset using case-insensitive string matching.



lxxi: Figure 5.1: Search Results for 'USB-C Charger'

5.3.2. Test Case: Sort Search Results by Price

xx: Table 5.3: Sort Search Results by Price

Date:	10 February 2026
System:	Bhao.pk
Objective:	Verify that results can be sorted by price ascending
Test ID:	2
Version:	1.0
Test Type:	Black Box - Functional

Input:	Search: "Iphone", Sort: "Price: Low to High"
Expected Result:	Products reordered with lowest price first.
Actual Result:	**Passed**

The sorting applies an ascending sort based on parsed price values using the useSearch hook.

lxxii: Figure 5.2: Results Sorted by Price (Low to High)

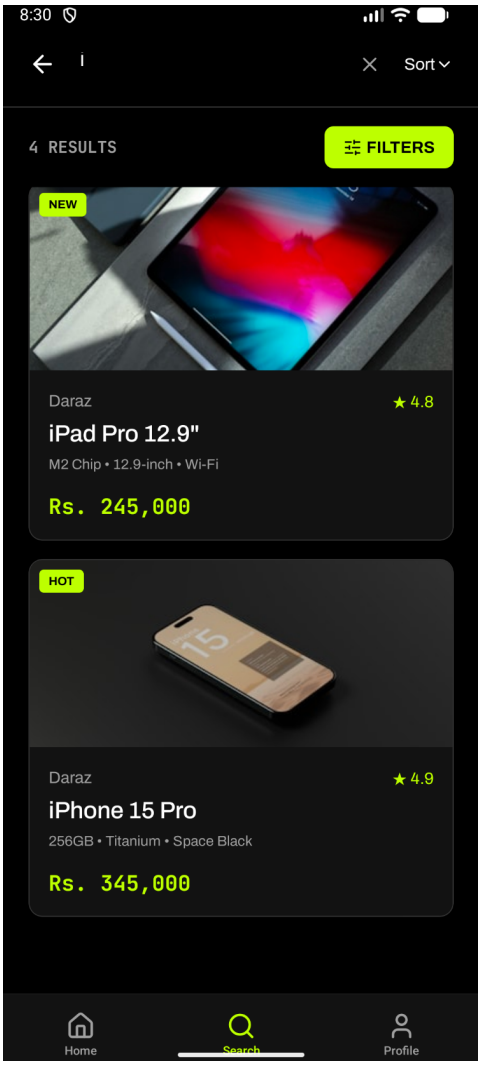
5.3.3. Test Case: Filter Results by Specific Store

xxi: Table 5.4: Filter Results by Store

Date:	10 February 2026
System:	Bhao.pk
Objective:	Verify that results can be filtered to a specific store

Test ID:	3
Version:	1.0
Test Type:	Black Box - Functional
Input:	Search: "Iphone", Store filter: "Daraz"
Expected Result:	Only Daraz products displayed. Telemart and Shophive items hidden.
Actual Result:	**Passed**

The store filter uses checkbox UI where each store can be toggled independently.



lxxiii: Figure 5.3: Results Filtered by Daraz Store

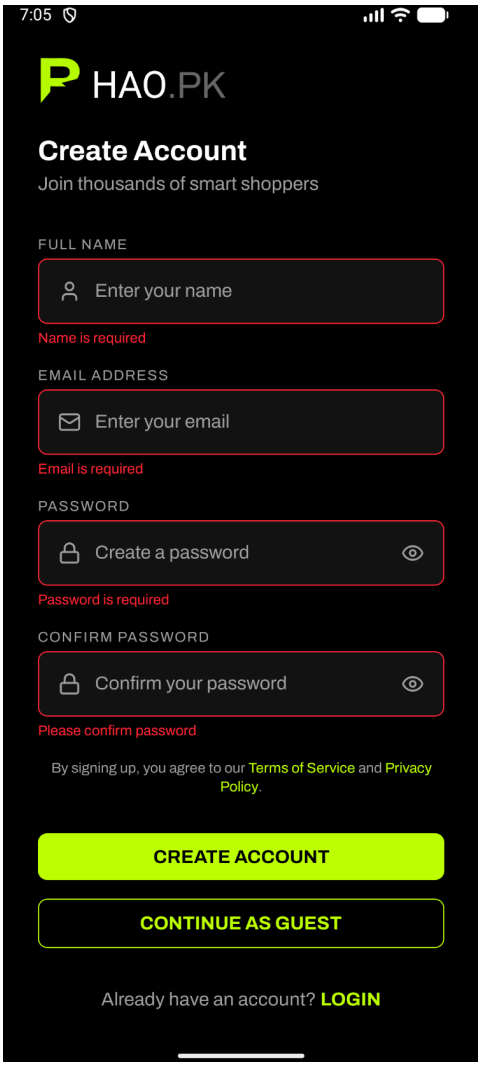
5.3.4. Test Case: User Sign Up

xxii: Table 5.5: User Sign Up

Date:	10 February 2026

System:	Bhao.pk
Objective:	Verify new user can create an account
Test ID:	4
Version:	1.0
Test Type:	Black Box - Functional
Input:	Name: "Test User", Email: "test@example.com", Password: "SecurePass123"
Expected Result:	Account created, user authenticated and redirected to home screen.
Actual Result:	**Passed**

Input validation uses a schema-based validation utility (src/utils/validation.ts) checking email format, password strength, and confirmation matching.



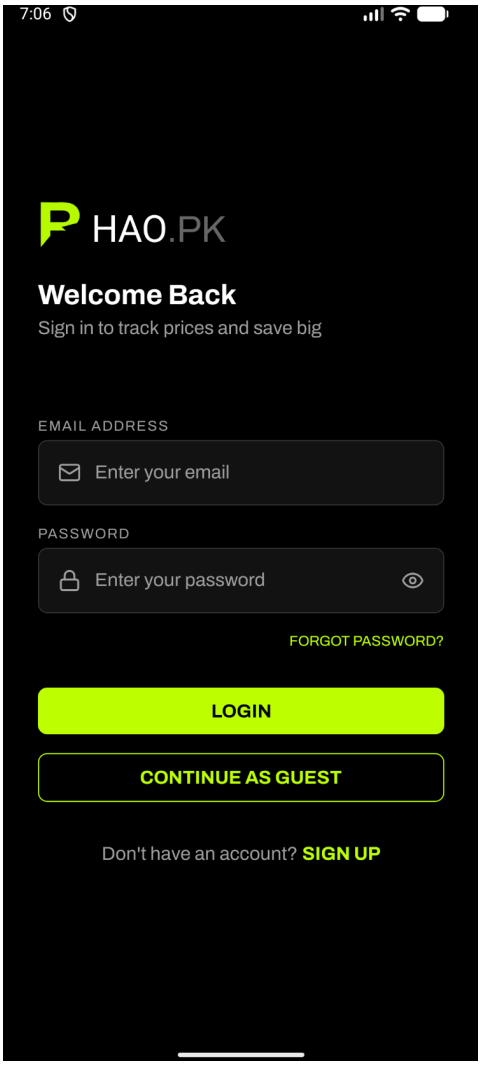
lxxiv: Figure 5.4: Sign Up Screen with Validation

5.3.5. Test Case: User Login

xxiii: Table 5.6: User Login

Date:	10 February 2026
System:	Bhao.pk
Objective:	Verify registered user can log in
Test ID:	5
Version:	1.0
Test Type:	Black Box - Functional
Input:	Email: "test@example.com", Password: "SecurePass123"
Expected Result:	User authenticated, session token stored, redirected to home screen.
Actual Result:	**Passed**

The AuthContext handles login, stores tokens in AsyncStorage, and restores sessions on app launch.



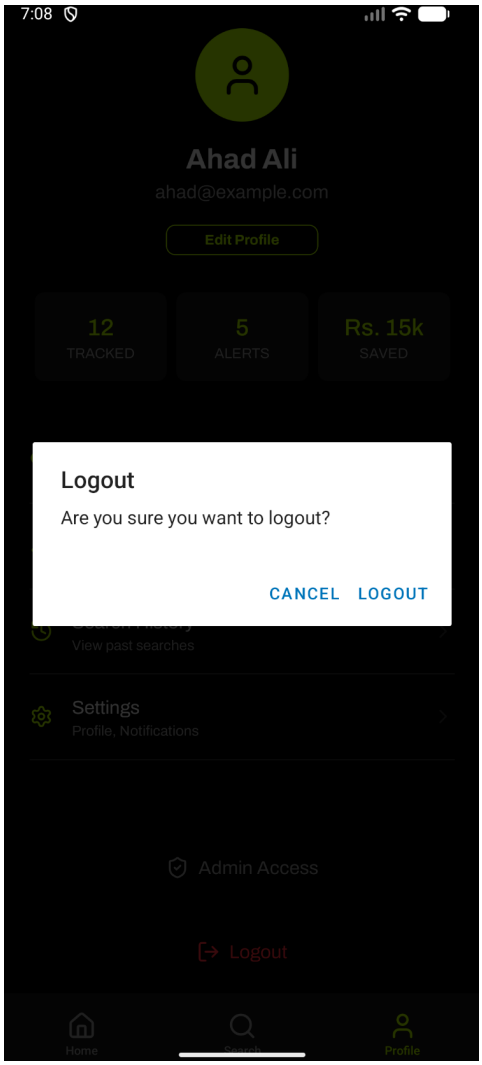
lxxv: Figure 5.5: Login Screen

5.3.6. Test Case: User Logout

xxiv: Table 5.7: User Logout

Date:	10 February 2026
System:	Bhao.pk
Objective:	Verify logged-in user can log out
Test ID:	6
Version:	1.0
Test Type:	Black Box - Functional
Input:	User navigates to Profile and taps Logout button.
Expected Result:	Confirmation dialog shown. Upon confirming, session cleared and redirected to Login.
Actual Result:	**Passed**

The logout uses a native Alert dialog with destructive style confirmation.



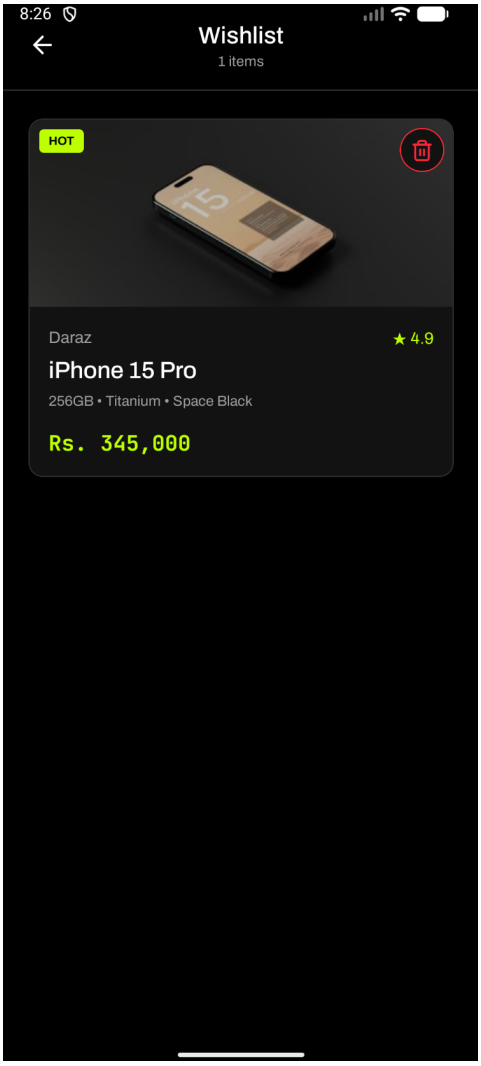
lxxvi: Figure 5.6: Logout Confirmation Dialog

5.3.7. Test Case: Add Product to Wishlist

xxv: Table 5.8: Add Product to Wishlist

Date:	10 February 2026
System:	Bhao.pk
Objective:	Verify user can add a product to their wishlist
Test ID:	7
Version:	1.0
Test Type:	Black Box - Functional
Input:	User taps heart icon on "iphone 15 Pro" product card.
Expected Result:	Heart icon fills, toast confirms addition, product appears in Wishlist screen.
Actual Result:	**Passed**

The useWishlist hook manages state and persists data to AsyncStorage across sessions.



lxxvii: Figure 5.7: Wishlist Screen with Saved Product

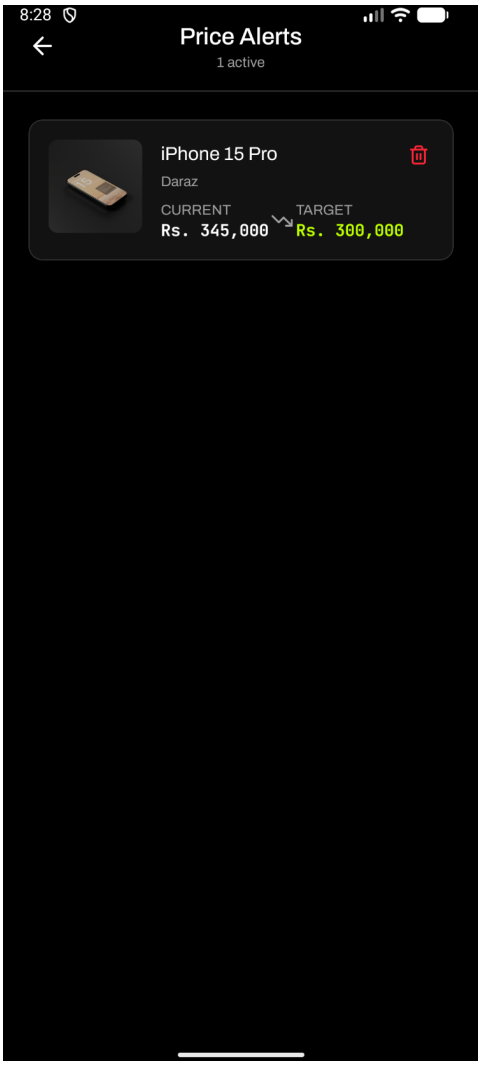
5.3.8. Test Case: Set Target Price Alert

xxvi: Table 5.9: Set Target Price Alert

Date:	10 February 2026
System:	Bhao.pk
Objective:	Verify user can set a price alert for a product
Test ID:	8
Version:	1.0
Test Type:	Black Box - Functional
Input:	Product: " iphone 15 Pro ", Target Price: PKR 300,000

Expected Result:	Alert created and visible in Alerts screen with product name, current price, and target.
Actual Result:	**Passed**

The AlertService handles CRUD operations for alerts through the ApiClient.



lxxviii: Figure 5.8: Alerts Management Screen

5.4. Test Results Summary

The following table summarizes all test results for this iteration.

Test ID	Functional Requirement	FR #	Status	Remarks
TC-01	Search Products by Keywords	FR-6	Passed	All stores returned results

TC-02	Sort Results by Price	FR-8	Passed	Ascending order verified
TC-03	Filter by Store	FR-9	Passed	Daraz filter tested
TC-04	User Sign Up	FR-13	Passed	Validation checks passed
TC-05	User Login	FR-14	Passed	Session persisted correctly
TC-06	User Logout	FR-15	Passed	Token cleared on logout
TC-07	Add to Wishlist	FR-21	Passed	Persisted in AsyncStorage
TC-08	Set Price Alert	FR-17	Passed	Alert visible in Alerts screen

xxvii: Table 5.10: Test Results Summary

All 8 test cases passed successfully, confirming that 30% of the functional requirements are correctly implemented. The remaining 70% will be tested in subsequent iterations as implementation progresses.