A project report on

NLP-Driven Intelligent Product Recommendation System

Bachelor of Technology in Computer Science and Engineering

by

Vemisetty Anshul (22BCE1308)

Akhil Tej Reddy (22BCE1619)

Dhanush Varma (22BCE1057)

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

CHENNAI

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

October 2025

ABSTRACT

The rapid expansion of e-commerce platforms has made product discovery increasingly complex, as users are often required to navigate through multiple filters and categories to locate the items they desire. Traditional product search systems rely heavily on structured input, such as dropdown filters or predefined categories, which limits their ability to interpret natural, human-like queries. To address this gap, this project proposes a "Personalized Product Finder using NLP-based Natural Query Processing and Flask Framework", a system designed to interpret free-text user input and deliver accurate, relevant product recommendations.

The system utilizes Natural Language Processing (NLP) techniques—implemented through a combination of keyword extraction and regular expressions (regex)—to analyze user queries and extract critical parameters such as brand, budget, product category, features, and intended use-case. These extracted attributes are then processed by a Flask-based backend that interfaces with an SQLite database containing product information. The system ranks and filters the products based on the degree of relevance to the extracted parameters, ensuring that the user receives the most suitable product recommendations in real time.

To ensure user and data management, the project integrates Flask-Login authentication for secure access and implements role-based authorization for users and admins. Admins can add, edit, or delete products dynamically, allowing the system to remain up-to-date without manual database modification. The frontend, developed using HTML, CSS (Bootstrap 5.3.2), and JavaScript (AJAX), offers a responsive and user-friendly interface that enables seamless interaction with the backend APIs.

Testing was conducted using Postman and manual test cases to verify the reliability, accuracy, and robustness of all functionalities—including NLP extraction, product filtering, authentication, and admin operations. Results demonstrate that the system performs efficiently, processing user queries and delivering recommendations with an average response time of less than two seconds.

The Personalized Product Finder successfully bridges the gap between human-like communication and data-driven product search. It provides a scalable foundation for future development, where advanced NLP models (such as spaCy or BERT) and machine learning–based recommendation techniques can be incorporated to further enhance accuracy and personalization. This system, therefore, represents a step toward intelligent, context-aware search engines that can transform the way users interact with digital marketplaces.

Table of Contents Structure

CONTENTS

CHAPTER 1 — INTRODUCTION

CHAPTER 2 — REQUIREMENTS & FEASIBILITY

CHAPTER 3 — SYSTEM DESIGN

CHAPTER 4 — IMPLEMENTATION

CHAPTER 5 — TESTING, RESULTS & DISCUSSION

CHAPTER 6 — CONCLUSION AND FUTURE SCOPE

CHAPTER 7 — REFERENCES

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

Acronym	Full Form
NLP	**Natural Language Processing**
API	**Application Programming Interface**
ORM	**Object Relational Mapping**
SQL	**Structured Query Language**
JSON	**JavaScript Object Notation**
HTML	**HyperText Markup Language**
CSS	**Cascading Style Sheets**
AJAX	**Asynchronous JavaScript and XML**
UI	**User Interface**
UX	**User Experience**
IDE	**Integrated Development Environment**
CLI	**Command Line Interface**
UML	**Unified Modelling Language**
VS Code	**Visual Studio Code**
OS	**Operating System**

CHAPTER 1 – INTRODUCTION

1.1 Introduction

In the modern era of digital commerce, users are increasingly dependent on intelligent systems to find products that match their needs, preferences, and budgets. Conventional e-commerce platforms typically rely on predefined filters or rule-based searches, requiring the user to manually select categories, brands, and price ranges. This process is time-consuming and often fails to capture the intent expressed in natural human language.

The Personalized Product Finder project addresses this problem by allowing users to describe their desired products in free-text form — for example, *"Show me gaming laptops under 80,000 with long battery life."* The system interprets such inputs using Natural Language Processing (NLP) techniques, specifically keyword and regular expression (regex) extraction, to understand user intent and match the input with an existing product database.

The system is built with a Flask (Python) backend, which exposes a RESTful API endpoint (/recommend) that accepts user queries and returns matching product suggestions in JSON format. A simple, responsive HTML-CSS- Bootstrap interface with AJAX-based JavaScript allows users to interact seamlessly with the backend.

This project demonstrates how natural language understanding can enhance product discovery, providing a foundation for future integration with more advanced AI-based models.

1.2 Background and Motivation

With the exponential growth of online marketplaces, the diversity of available products has expanded beyond what static filtering systems can manage. Users expect intuitive interfaces that understand natural language queries, similar to modern AI-driven assistants. However, implementing full-scale machine learning models can be resource-intensive and overkill for smaller systems.

This project is motivated by the need for a lightweight yet intelligent recommendation system that bridges this gap using fundamental NLP logic. By relying on simple Python-based keyword and regex extraction, the system delivers accurate results without the overhead of external NLP frameworks like spaCy or NLTK.

The Personalized Product Finder also serves an educational purpose: it helps students and developers understand the core interaction between NLP, databases, and backend APIs, forming a strong foundation for future work in intelligent recommendation systems and conversational commerce.

1.3 Problem Statement

Traditional recommendation systems depend on predefined filters or structured queries that restrict user freedom in expressing their needs. Users cannot simply "talk" to the system in natural language — they must conform to rigid dropdowns and input fields.

The core problem identified is:

To design and develop an NLP-based intelligent product recommendation system capable of interpreting user free-text queries and returning personalized product suggestions.

This involves building:

- A **Flask-based backend API** to process and respond to user queries.
- A **query-processing module** (nlp_utils.py) using keyword and regex-based extraction.
- A **product filtering mechanism** to match user preferences with database records.
- Role-based access control for **users and admins** to manage products dynamically.
- A **Postman-tested API** for verification, supported by automated functional testing through Selenium.

1.4 Objectives

The main objectives of the Personalized Product Finder project are as follows:

1. **To develop a Flask-based web application** capable of accepting natural language input and returning personalized product recommendations.
2. **To design and implement custom NLP utilities** using keyword extraction and regex matching for intent detection.
3. **To integrate a relational database (SQLite)** for efficient product and user data management using SQLAlchemy ORM.
4. **To provide role-based access control** using Flask-Login for secure user and admin authentication.
5. **To enable dynamic product management** by allowing admins to add, edit, or delete product records via web forms.
6. **To implement a lightweight front-end interface** using HTML, CSS (Bootstrap), and JavaScript for user interaction.
7. **To validate API endpoints and NLP accuracy** using Postman, Selenium, and performance testing tools.
8. **To create a scalable foundation** for future AI-based recommendation systems or voice-assisted product search.

1.5 Scope of the Project

The scope of this project encompasses the complete backend and partial frontend development of a personalized product search system with the following functionalities:

- **User Authentication:** Secure login and registration for both regular users and admins (role-based).
- **Product Management:** Admins can add, edit, and delete products via web dashboard.
- **NLP-based Recommendation:** Backend extracts information such as **budget**, **brand**, **category**, **use-case**, and **features** from user queries.
- **Product Catalog:** JSON-based dataset imported into SQLite using Python scripts.
- **API Access:** A /recommend POST endpoint for free-text queries.
- **Frontend:** Responsive Bootstrap interface with AJAX for dynamic query handling.
- **Testing:** API validation through Postman and automated functional testing using Selenium.

1.6 Relevance of the System

The relevance of this project lies in its real-world applicability and scalability. It demonstrates how even small-scale systems can implement AI-like behavior through logical NLP-driven techniques.

This system can easily be extended into an enterprise-grade application by integrating more sophisticated NLP frameworks, user profiling, and collaborative filtering algorithms.

From an educational and developmental standpoint, this project combines multiple domains:

- **Software Engineering:** Modular design, version control, and testing.
- **Database Management:** SQLite with ORM mapping (SQLAlchemy).
- **NLP Concepts:** Keyword extraction and regex-based feature detection.
- **Web Development:** RESTful API design with Flask.
- **Testing and Automation:** Postman and Selenium integration.

The project demonstrates the synergy between software engineering and intelligent systems, showing how structured design methodologies can result in a system that is functional, user-friendly, and intelligent.

1.7 Limitations of the Project

While the Personalized Product Finder meets its primary objectives, several limitations remain that define areas for enhancement:

1. **Static Dataset:** Product data is limited to the predefined products.json file. Real-time updates or external API integrations are not yet supported.

2. **Limited Scalability:** The Flask application is configured for local development; it does not include cloud deployment or horizontal scaling.
3. **Single-Language Support:** Currently supports only English queries; no multilingual tokenization is implemented.
4. **No Machine Learning Model:** Recommendations are logic-driven, not predictive; personalization does not evolve from user behaviour.

CHAPTER 2 – REQUIREMENTS & FEASIBILITY

2.1 Functional Requirements

Functional requirements define what the system should do — the specific behavior, features, and logic that make the Personalized Product Finder operational.
These are derived from the objectives defined in Chapter 1 and describe both user-facing and administrative functionalities.

User-Related Functionalities:

ID	Requirement Description
FR-01	**User Registration:** The system shall allow new users to register by providing username, password, email, and optional contact details.
FR-02	**User Login:** Registered users shall be able to log in using secure authentication via Flask-Login.
FR-03	**Password Management:** Users shall be able to update or reset their passwords after authentication.
FR-04	**Query Submission:** Users shall be able to enter free-text product queries such as “show me Apple phones under 50 000”.
FR-05	**NLP Query Processing:** The system shall extract parameters (budget, brand, category, use case, and features) from the user’s text using regex and keyword extraction.
FR-06	**Product Recommendation:** The system shall match extracted parameters against the database and return the most relevant products.
FR-07	**Result Display:** The recommended products shall be shown in a card-based interface containing name, price, brand, and category.
FR-08	**Bookmarking/Re-query (optional):** The user may bookmark selected products or re-run previous queries for comparison.

Table 1: User Functionalities

Admin-Related Functionalities:

ID	Requirement Description
FR-09	**Admin Registration:** Admin users shall register only using a valid AdminID (e.g., ADMIN001).
FR-10	**Admin Dashboard:** The admin shall have access to a dashboard showing all products with options to search, edit, or delete records.
FR-11	**Add Product:** The admin shall add new products by specifying name, category, sub-category, brand, price, and features.
FR-12	**Edit/Delete Product:** The admin shall modify or remove product details through secured routes.
FR-13	**Data Import Utility:** The system shall allow bulk import of products from a products.json file.

Table 2: Admin Functionalities

System-Level Functionalities:

ID	Requirement Description
FR-14	**Database Handling:** All data shall be stored and retrieved using SQLite via SQLAlchemy ORM.
FR-15	**Session Management:** The system shall manage user sessions securely using Flask-Login.
FR-16	**API Interface:** The /recommend endpoint shall accept JSON queries and return JSON responses for easy external integration.
FR-17	**Health Check:** The /health route shall confirm server status with a simple JSON response { "status": "ok" }.

Table 3: User Functionalities

2.2 Non-Functional Requirements

Non-functional requirements define **how** the system performs rather than **what** it performs.
They focus on system qualities such as performance, usability, reliability, and security.

Category	Requirement Description
Performance	System should process and respond to user queries within **3 seconds** under normal load.
Scalability	The architecture should allow easy migration to MySQL or PostgreSQL if dataset size increases.
Security	Passwords shall be hashed using Werkzeug; user sessions managed securely with Flask-Login.
Usability	Interface should be responsive and intuitive using Bootstrap 5; forms and error messages should be user-friendly.
Reliability	Application should recover gracefully from errors (e.g., invalid inputs, missing data).
Maintainability	Code should follow modular design (separate files for models, routes, NLP logic) to ease debugging and future upgrades.
Portability	The system should run on any machine with Python ≥ 3.10 and minimal configuration.
Testability	Endpoints must be verifiable via Postman and Selenium; unit tests should be possible on individual modules.
Data Integrity	Product and user data consistency maintained through SQLAlchemy ORM validation.
Compliance	Follow good software-engineering practices for naming conventions, documentation, and testing standards.

Table 4: Non Functionalities

2.3 Constraints

Constraints outline the practical limits under which the project was developed.

Technological Constraints

- **Backend Framework:** Restricted to Flask 3.1.2 — no external machine-learning libraries allowed.
- **Database:** SQLite (file-based) — chosen for simplicity over server-based databases.
- **NLP Methodology:** Limited to regex and keyword extraction; no advanced NLP models.
- **Environment:** Local development mode (Flask debug); no production deployment.

Budget Constraints

- Academic project with zero external funding; uses only free and open-source tools.
- Hosting and external APIs excluded to avoid cost.
- Hardware limited to standard laboratory/college systems.

Time Constraints

- Project duration: approximately 12 weeks (August – October 2025).
- Development divided into planning, design, coding, testing, and documentation phases.
- Limited testing cycles due to academic schedule.

2.4 Feasibility Study

A feasibility study assesses the project's practicality with respect to technology, economics, and operations.

2.4.1 Technical Feasibility

The project is technically feasible because all components are based on accessible and well-supported technologies:

- **Programming Language:** Python — easy to learn and integrates well with NLP and web frameworks.
- **Framework:** Flask provides lightweight, modular design suitable for academic prototypes.
- **Database:** SQLite offers fast setup and zero-configuration storage for small to medium datasets.
- **Tools & IDEs:** Visual Studio Code, Postman, and Google Colab used for coding and testing.
- **Dependencies:** All required libraries (Flask, SQLAlchemy, Flask-Login, etc.) are freely available via pip.
- **Hardware Requirements:** Standard computer with at least 4 GB RAM and stable Python environment.

Thus, from a technical standpoint, all resources required for system implementation were available and sufficient.

2.4.2 Economic Feasibility

The project incurs **no monetary cost** beyond basic infrastructure:

- **Software Cost:** All technologies used are open-source (Flask, SQLite, Bootstrap, Python libraries).

- **Hardware Cost:** Utilized existing laboratory and personal computers.
- **Manpower:** Two team members contributed equal development effort (design, coding, documentation).
- **Maintenance Cost:** Negligible, since updates involve editing Python or JSON files.

Therefore, the system is economically feasible for academic and small-business deployment contexts.

2.4.3 Operational Feasibility

Operational feasibility evaluates whether the developed system functions effectively within the user environment.

- The application can be launched with a single command (python app.py) in a local environment.
- End-users interact through a web interface or Postman, without needing technical knowledge.
- The admin dashboard ensures easy management of the product catalogue.
- The NLP module operates transparently, interpreting natural queries and providing meaningful results.
- Minimal training is required for end-users; basic familiarity with typing queries is sufficient.

Hence, operational feasibility is high, as the system's usability, accessibility, and simplicity align with user expectations.

CHAPTER 3 — SYSTEM DESIGN

3.1 System Architecture

Purpose: Present the overall architecture of the Personalized Product Finder system — how components interact, where processing occurs, and how data flows end-to-end.

3.1.1 Architecture overview

The system follows a lightweight **three-tier architecture**:

1. **Presentation Layer (Client)**
    - Browser-based UI (HTML/Bootstrap/Jinja2) and optional API client (Postman).

- Responsible for accepting free-text queries from users and displaying the returned product cards.

2. **Application / Logic Layer (Backend)**
    - Flask application (APIController) exposes REST endpoints (/recommend, /health, admin routes).
    - Contains core modules:
        - **NLP Processor (nlp_utils.py)** — regex and keyword extraction logic.
        - **Recommendation Engine (product_filter.py)** — filter, score, and rank products.
        - **Auth module** — Flask-Login and Werkzeug password hashing.
        - **Admin controllers** — add/edit/delete product functionality.
3. **Data Layer (Persistence)**
    - SQLite DB (via SQLAlchemy) stores Product, User, AdminID tables.
    - Initial data seeded from products.json; import_products_json.py and seed_db.py support seeding.

Communication: Client ↔ (HTTPS/HTTP) ↔ Flask API ↔ (ORM/SQL) ↔ SQLite. Optionally, the API may call an external NLP service if configured, but the current design uses local nlp_utils.py.

3.1.2 Architectural diagram

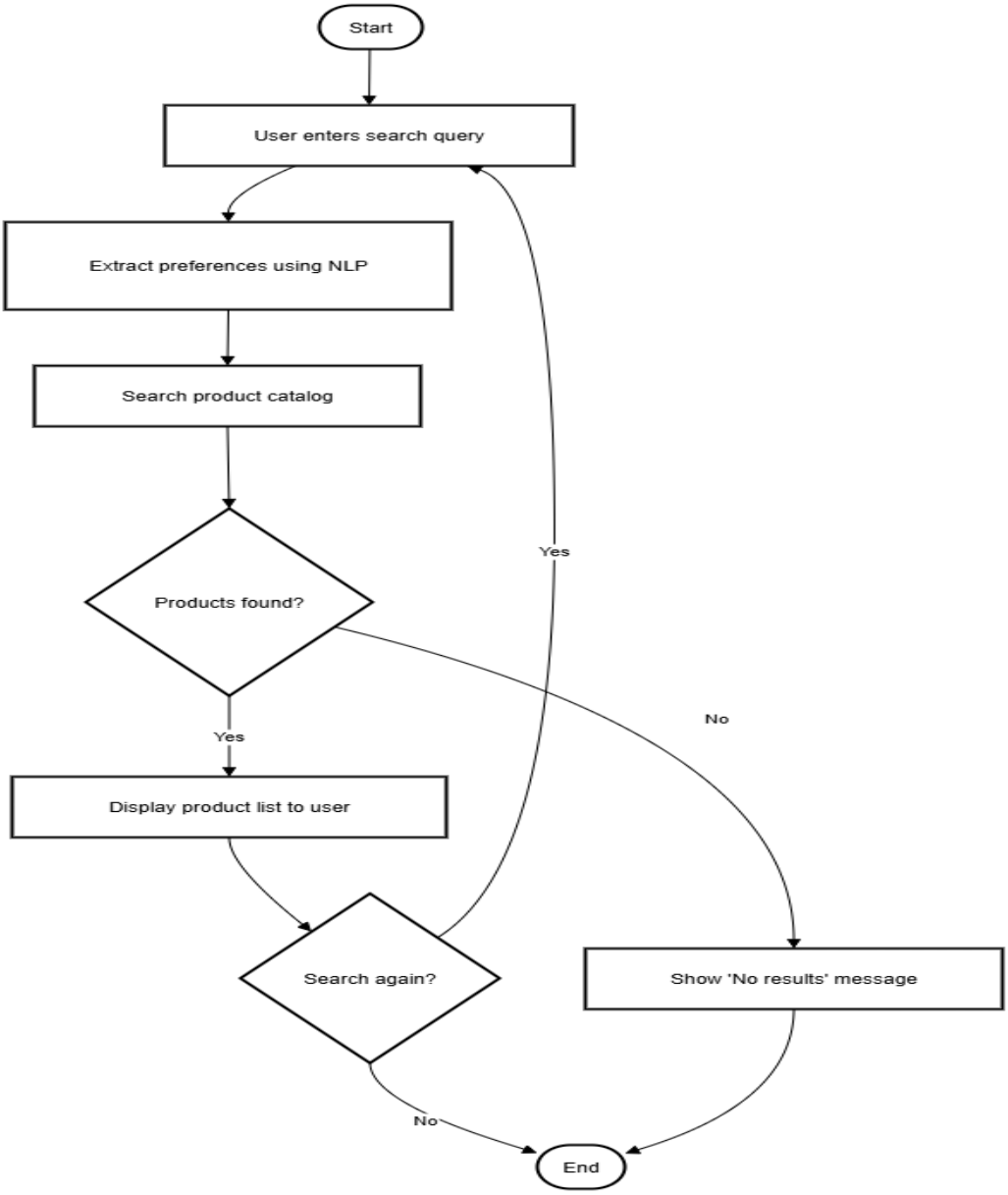


Figure 3.1: System Architecture

3.2 Data Flow Diagrams (Levels 0–3)

Purpose: Show decomposition of data movement through the system from high level (Level 0) to detailed internal steps (Level 3).

Note: You have already drawn DFDs in StarUML. Below are textual descriptions for each level that match those diagrams and can be used as captions/explanations.

Level 0 DFD — Context diagram

Single process bubble: Personalized Product Finder System

External entities:

- **User** — submits queries, views results, bookmarks items.
- **Admin** — uploads/updates product catalog.
- **Optional External NLP Service** — fallback or replacement for local NLP.

Primary Data Stores: Product Catalog (products.json / DB)

Major flows:

- User → (query text) → System → (recommended products) → User
- Admin → (catalog upload) → System → (catalog stored) → Product Catalog

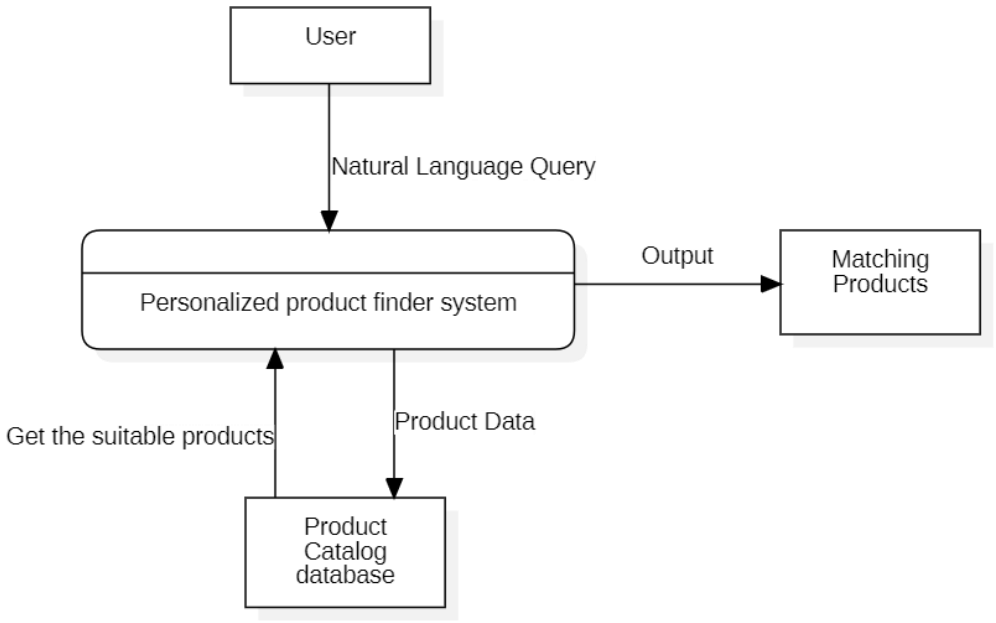


Figure 3.2: DFD Level 0 — System as single process showing User and Admin interactions.

Level 1 DFD — Top-level decomposition

Processes (decomposed):

1. **Enter Query** — UI accepts user free-text.
2. **Process Query (NLP)** — extract budget, brand, features, sub-category.
3. **Match Products** — apply filters against Product Catalog.
4. **Return Results** — package and return top ranked products.
5. **Manage Catalog** — admin add/update/delete operations.

Data stores:

- Product Catalog (DB)
- User Preferences / Bookmarks (if used)

Flows:

- Query text → NLP Processor → Extracted Parameters → Matching → Candidate list → Ranking → JSON Response

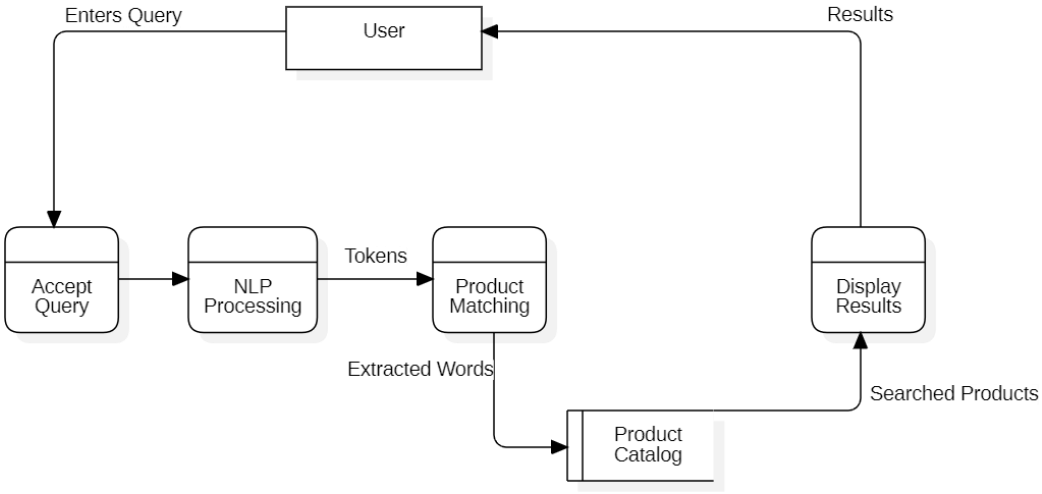


Figure 3.3: DFD Level 1 — major functional components and flows.

Level 2 DFD — Detailed decomposition of Process Query and Match Products

Process Query sub-processes:

- **Tokenize & Normalize** — lowercasing, punctuation removal.
- **Budget Parsing** — regex for under X, below X, between X and Y.
- **Brand Detection** — exact and fuzzy matching against known_brands.
- **Feature Extraction** — scan for known_features.
- **Use-Case Mapping** — map phrases like “for office” → use_case.

Match Products sub-processes:

- **Filter by Category/Sub-category** — strict or fallback matching.
- **Filter by Budget** — price <= budget.
- **Filter by Brand** — if brand specified.

- **Scoring** — brand match (+2), feature/use_case matches (+1 each), price distance (+1 if price ≤ budget - threshold).
- **Sort & Truncate** — return top N results.

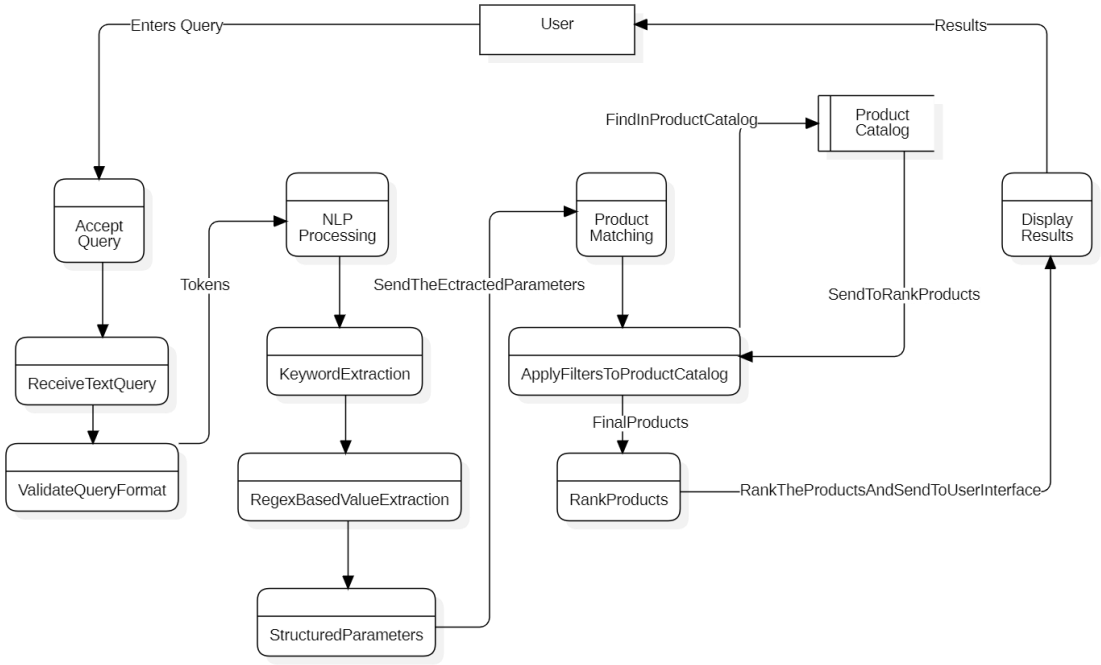


Figure 3.4: DFD Level 2 — expands NLP processing and matching internals.

Level 3 DFD — Internal data transformations and storage

Examples of Level 3 detail for NLP Processing:

- **BudgetRegex Module → budget_value**
- **BrandLookup Module → brand_tag**
- **FeatureMapper → feature_list**
- **ParamAssembler → params dict**

Examples of Level 3 detail for Matching:

- For each product in catalog: compute score = brand_score + feature_matches + use_case_matches + price_bonus → push to candidates → sort.

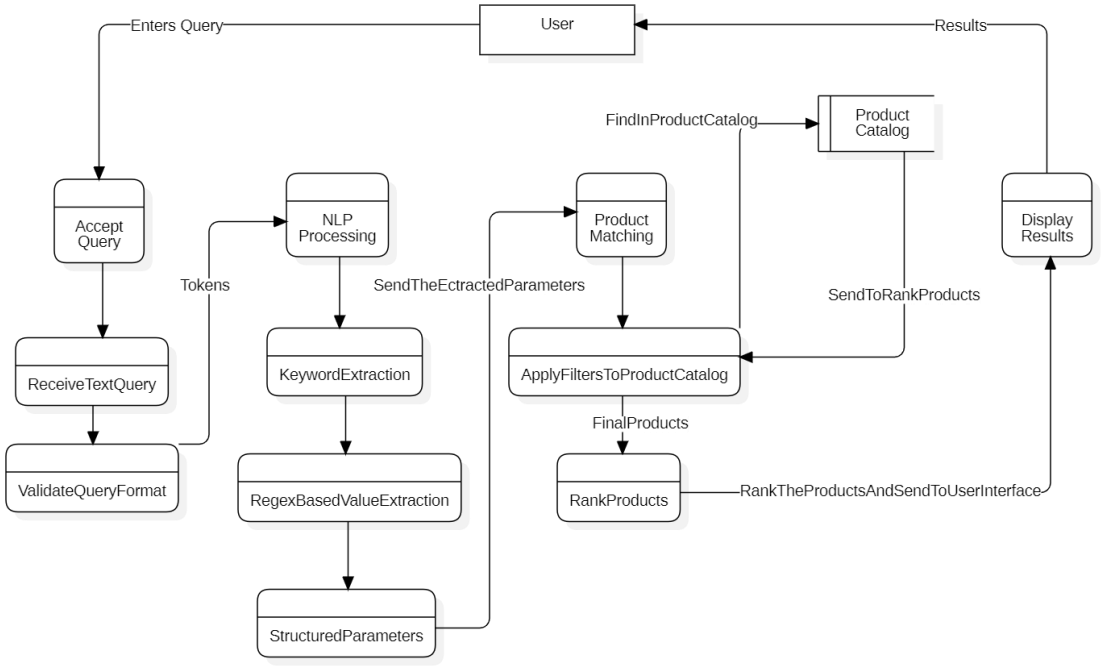


Figure 3.5: DFD Level 3 — shows internal activities, decision nodes, and object flows for a single query processing pipeline.

3.3 Use Case Diagram

Purpose: Identify actors and their goals (use cases) — clarifies functional requirements visually.

3.3.1 Actors

- **User** — primary consumer of the system (search, refine, bookmark).
- **Admin** — manages product catalog (upload, edit, delete).
- **System (or NLP Engine)** — external actor if representing an external NLP service.

3.3.2 Use cases

- **Enter Product Query**
- **Process Query via NLP** (included in Enter Product Query)
- **View Matching Products**
- **Refine / Re-query**
- **Bookmark Product**
- **Register / Login / Logout**

- **Upload / Update Catalog**
- **Manage Users (Admin)**

3.3.3 Diagram description (walkthrough)

1. **User** interacts with **Enter Product Query**.
2. **Enter Product Query** invokes **Process Query via NLP** (system-internal use case).
3. **Process Query via NLP** produces structured parameters for **View Matching Products**.
4. **User** may choose **Refine / Re-query** or **Bookmark Product**.
5. **Admin** interacts with **Upload/Update Catalog**, which modifies the Product Catalog data store.

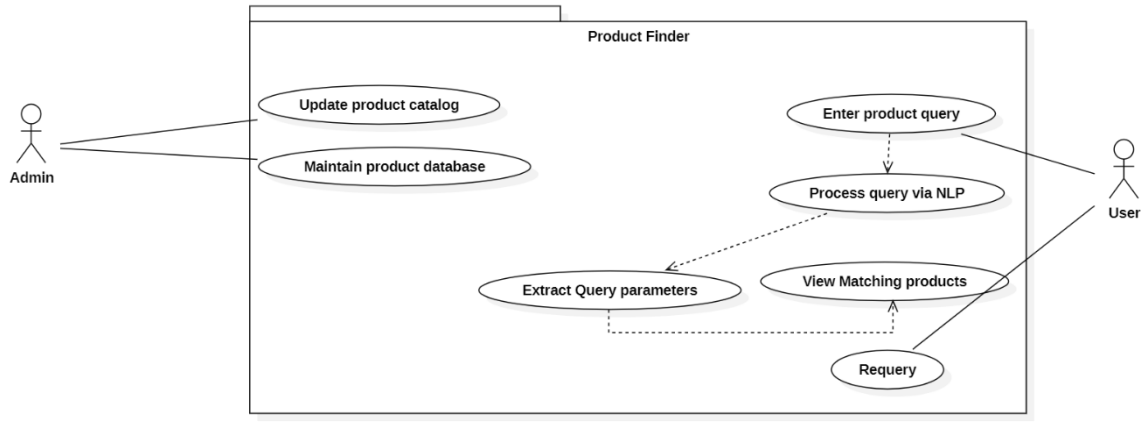


Figure 3.6: Use Case Diagram

3.4 Class Diagram

Purpose: Show static structure — classes, attributes, methods, and relationships used in the backend.

You drew this in StarUML; below is a precise textual mapping to the classes you modeled.

3.4.1 Main classes and members

Product

- Attributes:
    - id: Integer (PK)
    - name: String
    - category: String

    - sub_category: String
    - brand: String
    - price: Integer
    - features: String (comma-separated)
    - use_case: String (comma-separated)
    - size: String (optional)
- Methods:
    - to_dict(): dict
    - matches(params): bool (used by RecommendationEngine)

User (inherits UserMixin)

- Attributes:
    - id: Integer (PK)
    - username: String
    - password_hash: String
    - role: String (user or admin)
    - email: String
    - mobile_number: String (optional)
- Methods:
    - check_password(plain): bool
    - set_password(plain): void

AdminID

- Attributes:
    - id: Integer
    - admin_code: String (e.g., ADMIN001)

APIController (not a DB class, but modeled for design)

- Methods:
    - post_recommend(query_json): Response
    - post_bookmark()
    - admin_upload_catalog()

NLPProcessor

- Methods:
    - extract_params(query_text): dict — returns { budget, brand, features, use_cases, sub_category }

RecommendationEngine

- Methods:
    - match_products(params): list[Product] — uses Product.to_dict and scoring.

DataAccess / CatalogDAO

- Methods:
    - get_all_products(): list[Product]
    - get_product_by_id(id): Product
    - upsert_product(product_data): bool
    - delete_product(id): bool

3.4.2 Relationships

- APIController uses NLPProcessor and RecommendationEngine.
- RecommendationEngine depends on DataAccess / Product instances.
- Admin (actor) communicates with APIController admin routes.

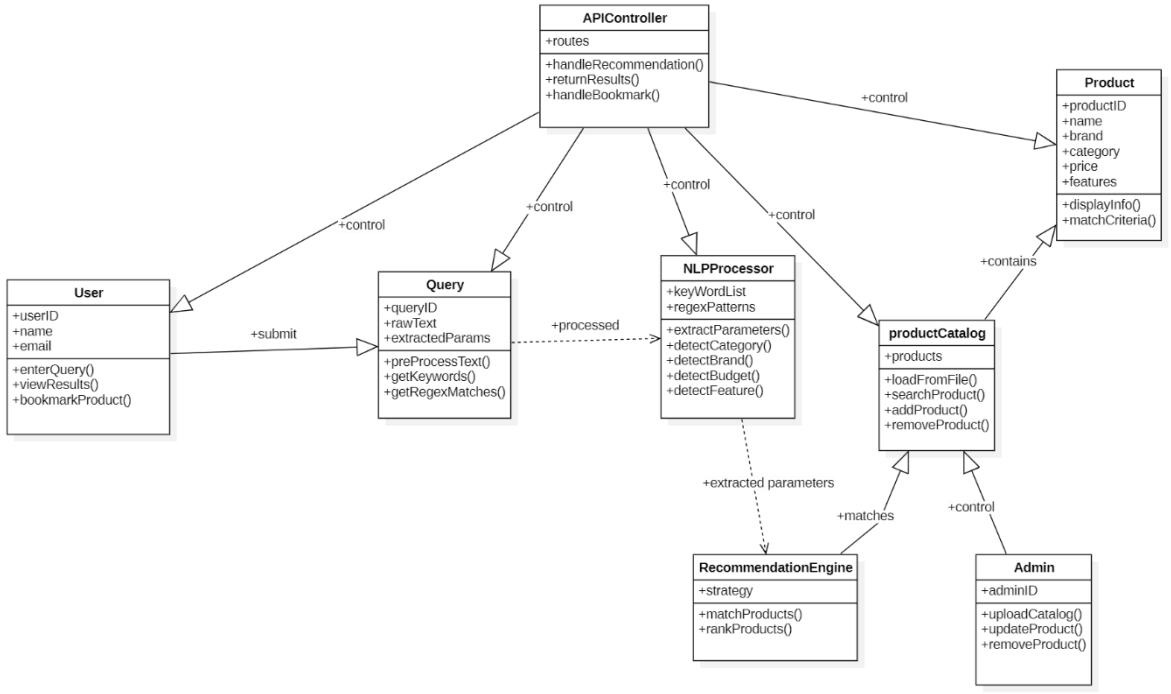


Figure 3.7: Class Diagram

3.5 Activity Diagram

Purpose: Model workflows of primary system activities (non-swimlane and swimlane versions). You created both in StarUML.

3.5.1 Non-Swimlane activity diagram — Search flow (textual)

Flow:

1. **Start** → *Enter Query*
2. *Submit Query* → *NLP Processing*
3. *Extract Parameters* → decision: **Results Found?**
    - Yes → *Display Results* → *Refine/Bookmark?* → loop or *End Session*
    - No → *Show Alternatives* → *Refine Query*
4. **End**

3.5.2 Swimlane activity diagram — Actors: User / System / Admin

User lane: Enter Query → Submit Query → View Results → Choose Refine or Bookmark → End.
System lane: Receive Query → NLP → Match → Rank → Return Results.
Admin lane: Upload Catalog → Validate → Apply Update or Rollback.

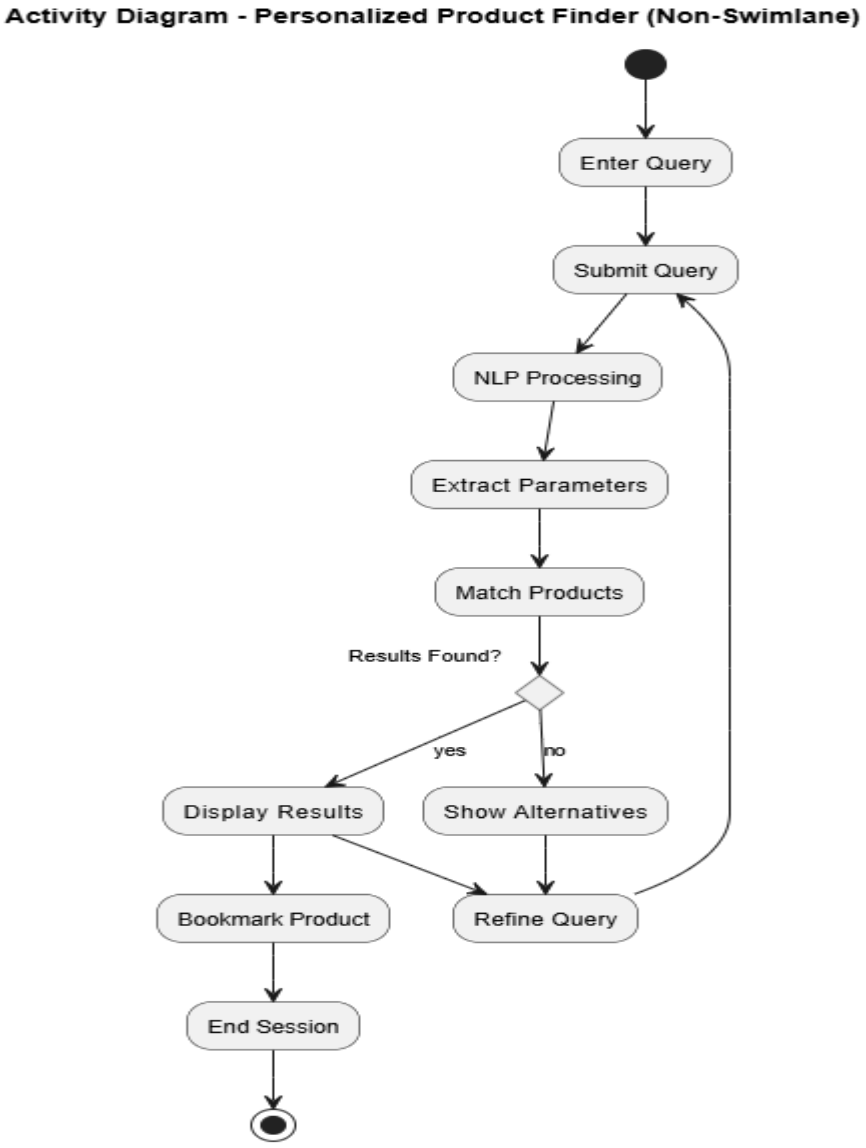


Figure 3.8: Activity Diagram (non-swimlane) — sequence of actions for search.

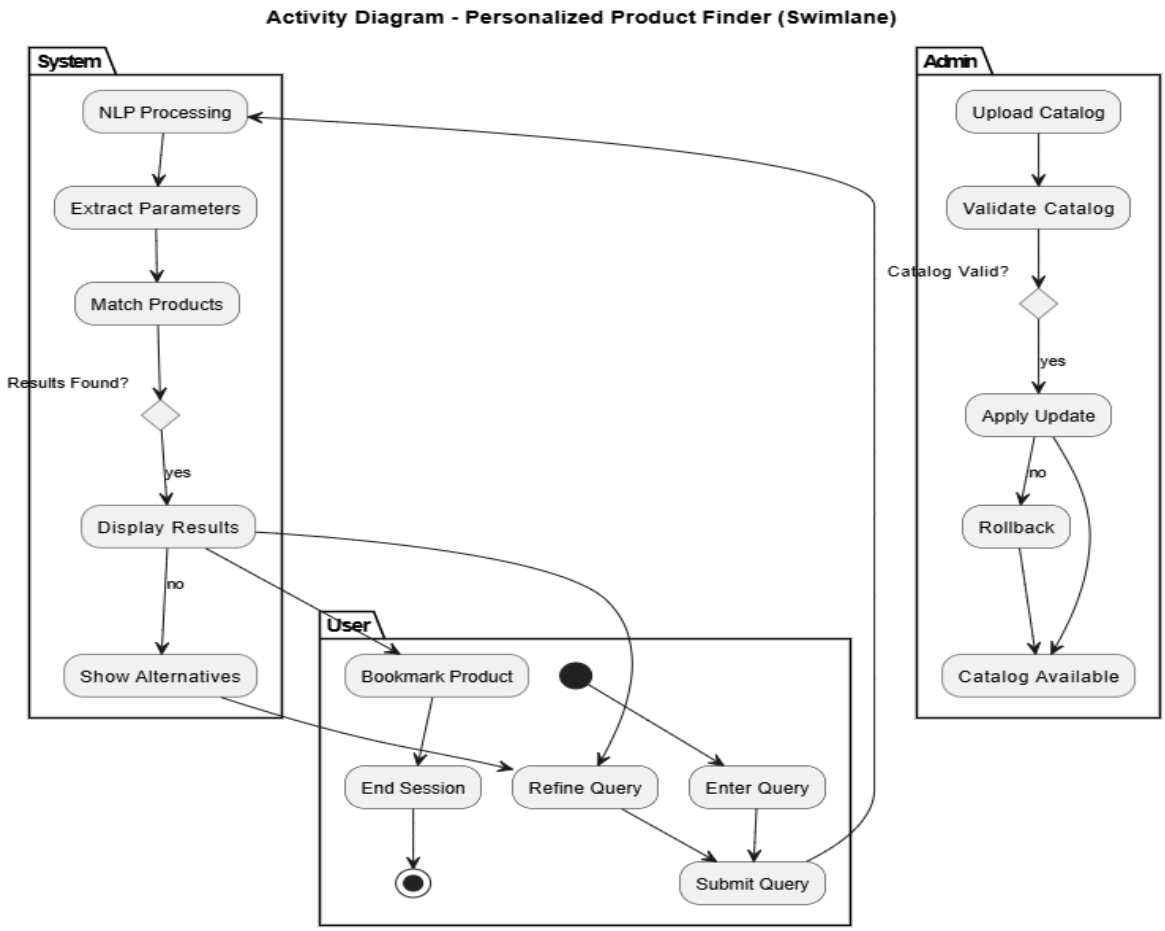


Figure 3.9: Activity Diagram (swimlane) — shows responsibility partitioning between User, System, and Admin.

3.6 Sequence Diagrams

Purpose: Show time-ordered interactions for two main scenarios. You drew two sequence diagrams in StarUML: (1) Search Products via NLP and (2) Admin Uploads/Updates Catalog.

Below are detailed message lists matching your diagrams.

3.6.1 Sequence: Search Products via NLP

Lifelines (left → right):

- User (actor)
- Client/UI (boundary)
- APIController (control)
- Query (entity/value object)
- NLPProcessor (control)

- RecommendationEngine (control)
- ProductCatalog (entity/DB)

Message flow:

1. User → Client/UI: enterQuery(text)
2. Client/UI → APIController: POST /recommend(queryText)
3. APIController → Query: new(queryText)
4. APIController → NLPProcessor: extractParameters(query)
5. NLPProcessor → APIController: params
6. APIController → RecommendationEngine: matchProducts(params)
7. RecommendationEngine → ProductCatalog: search(params)
8. ProductCatalog → RecommendationEngine: productCandidates[]
9. RecommendationEngine (self): rankProducts(productCandidates)
10. RecommendationEngine → APIController: rankedProducts[]
11. APIController → Client/UI: 200 OK + JSON(rankedProducts)
12. Client/UI → User: showResults()

Combined fragments:

- alt [invalid query] → APIController → Client/UI: 400 Bad Request
- alt [no matches] → APIController → Client/UI: 200 OK + []
- loop for pagination: RecommendationEngine ↔ ProductCatalog: search(nextPage)

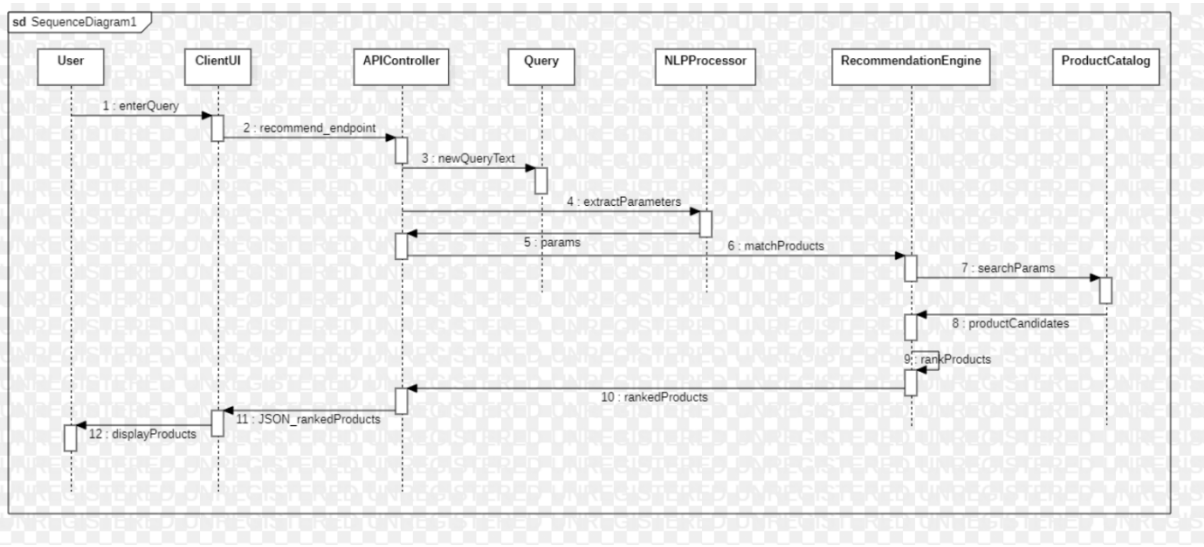


Figure 3.10: Sequence Diagram (User)

3.6.2 Sequence: Admin Uploads/Updates Catalog

Lifelines:

- Admin
- Admin UI
- APIController
- Validator
- ProductCatalog
- FileStorage (artifact)

Message flow:

1. Admin → Admin UI: chooseFile(products.json)
2. Admin UI → APIController: POST /catalog/upload(file)
3. APIController → Validator: validateSchema(file)
4. Validator → APIController: validationResult(ok | errors[])
5. alt [validation failed]: APIController → Admin UI: 400 + errors (END)
6. alt [validation ok]: APIController → ProductCatalog: parseAndStage(file)
7. loop [for each product]: ProductCatalog: upsert(product)
8. ProductCatalog → FileStorage: writeAll(products)
9. FileStorage → ProductCatalog: writeResult(success | failure)
10. alt [write failure]: ProductCatalog → APIController: updateStatus(failed) → APIController → Admin UI: 500 error
11. ProductCatalog → APIController: updateSummary(added, updated, skipped)
12. APIController → Admin UI: 200 OK + summary
13. Admin UI → Admin: showUploadSummary()

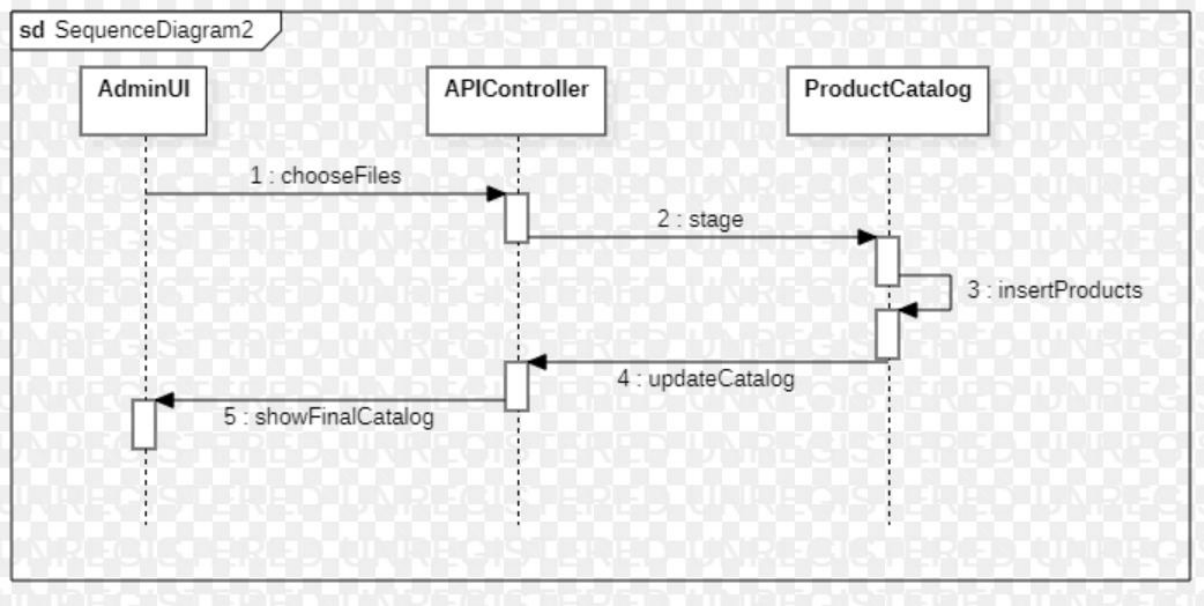


Figure 3.11: Sequence Diagram (Admin)

3.7 Collaboration (Communication) Diagrams

Purpose: Show object relationships and message sequence using numbered interactions (same behavior as sequence diagrams but emphasizing links).

You drew two collaboration diagrams corresponding to the sequence flows. Below are the object nodes and numbered messages.

3.7.1 Collaboration: Search Products via NLP — Nodes & Messages

Nodes: :User, :Client/UI, :APIController, :Query, :NLPProcessor, :RecommendationEngine, :ProductCatalog

Messages (numbered):

1. User → Client/UI: enterQuery(text)
2. Client/UI → APIController: POST /recommend(queryText)
3. APIController → Query: new(queryText)
4. APIController → NLPProcessor: extractParameters(query)
5. NLPProcessor → APIController: params
6. APIController → RecommendationEngine: matchProducts(params)
7. RecommendationEngine → ProductCatalog: search(params)
8. ProductCatalog → RecommendationEngine: productCandidates[]
9. RecommendationEngine → RecommendationEngine: rankProducts(productCandidates)

10. RecommendationEngine → APIController: rankedProducts[]
11. APIController → Client/UI: JSON(rankedProducts)
12. Client/UI → User: showResults()

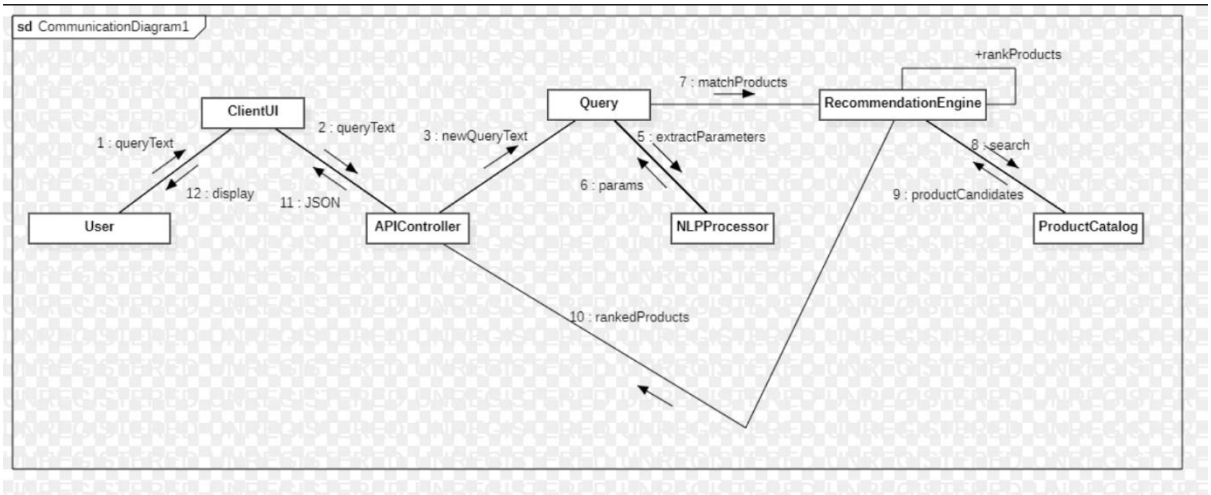


Figure 3.12: Collaboration Diagram — Search flow.

3.7.2 Collaboration: Admin Uploads Catalog — Nodes & Messages

Nodes: :Admin, :Admin UI, :APIController, :Validator, :ProductCatalog, :FileStorage

Messages (numbered):

1. Admin → Admin UI: chooseFile(products.json)
2. Admin UI → APIController: POST /catalog/upload(file)
3. APIController → Validator: validateSchema(file)
4. Validator → APIController: validationResult(ok/errors)
5. APIController → ProductCatalog: parseAndStage(file) (if ok)
6. ProductCatalog → ProductCatalog: upsert(product) (loop)
7. ProductCatalog → FileStorage: writeAll(products)
8. FileStorage → ProductCatalog: writeResult(success)
9. ProductCatalog → APIController: updateSummary(added, updated, skipped)
10. APIController → Admin UI: 200 OK + summary

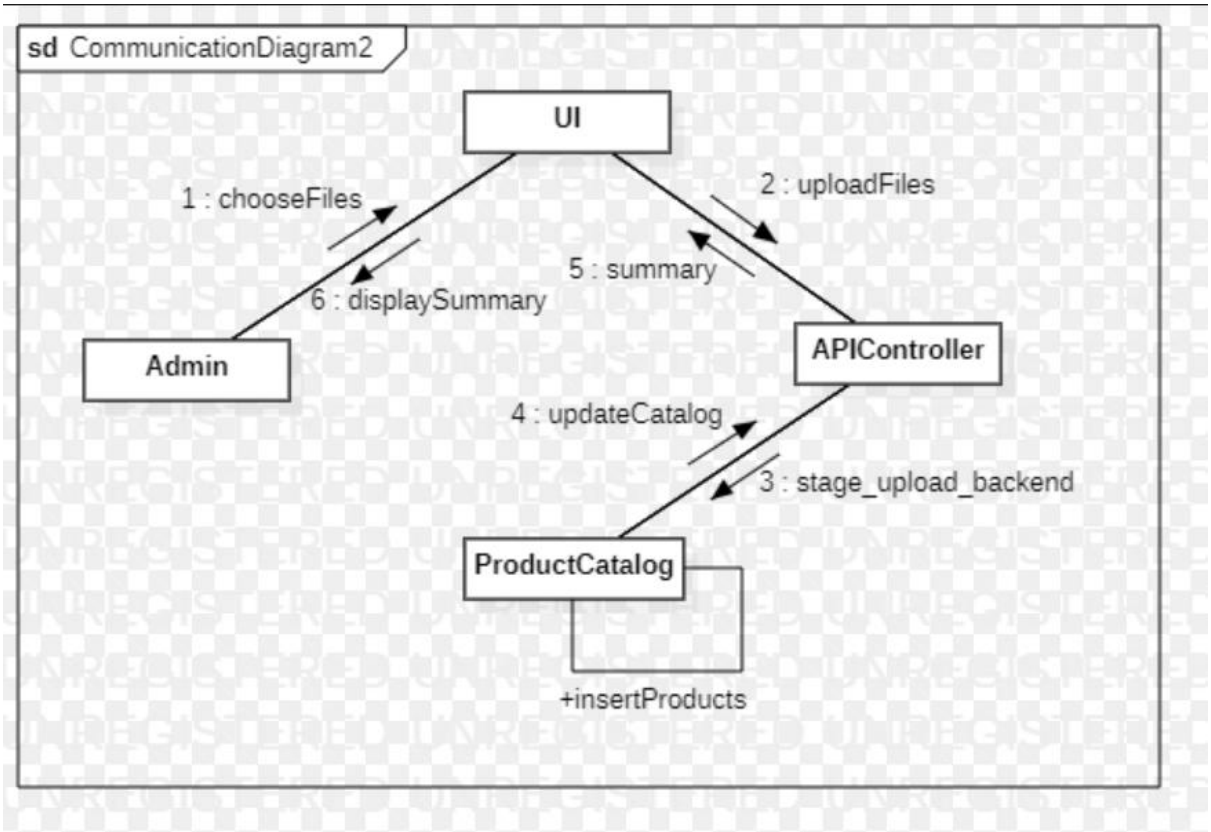


Figure 3.13: Collaboration Diagram — Admin upload flow.

3.8 Component Diagram

Purpose: Show high-level components (deployable units / modules), provided/required interfaces and relationships.

3.8.1 Components in diagram

- **Web UI / Client** — interacts via IRecommendAPI.
- **API Controller (Flask App)** — exposes IRecommendAPI.
- **NLP Processor** — provides INLP.
- **Recommendation Engine** — provides IRecommendation.
- **Product Catalog / Data Access** — provides IDataAccess.
- **Admin Panel** — provides IAdminAPI.
- **Auth & Config** — cross-cutting module.
- **Monitoring & Logging** — optional observability module.

3.8.2 Interfaces and connections

- Web UI → APIController (IRecommendAPI)
- APIController → NLPProcessor (INLP)

- APIController → RecommendationEngine (IRecommendation)
- RecommendationEngine → ProductCatalog (IDataAccess)
- Admin Panel → ProductCatalog & APIController (IAdminAPI)

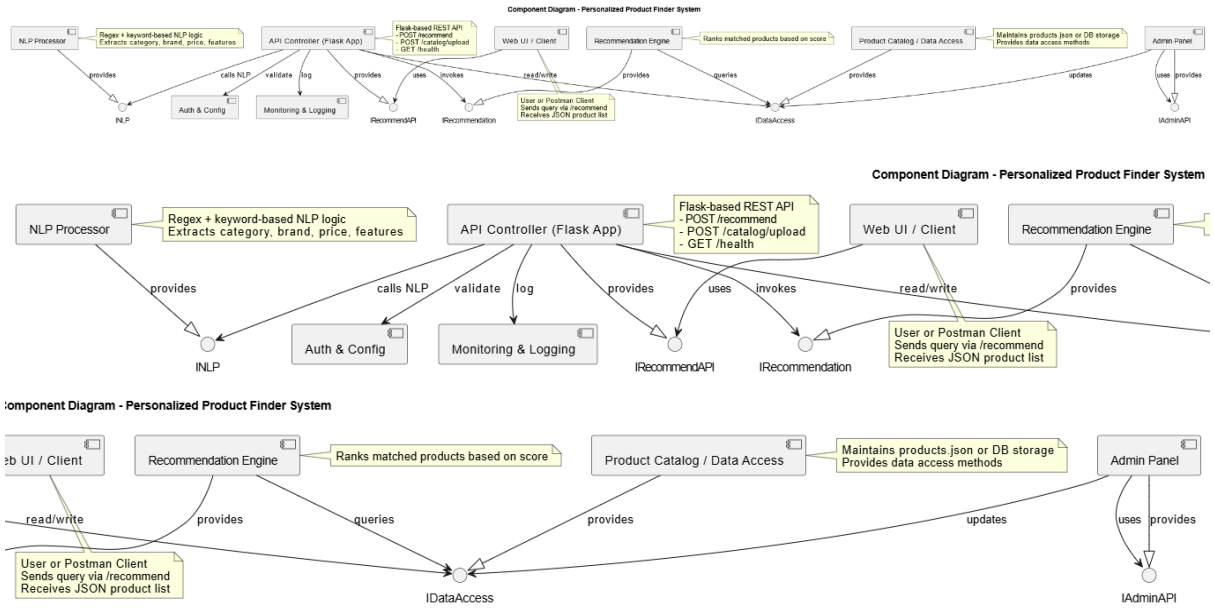


Figure 3.14: Component Diagram — shows modules, provided/required interfaces, and dependencies.

3.9 Package Diagram

Purpose: Show logical grouping of modules in the codebase (packages) and package-level dependencies.

3.9.1 Packages and contents

- **ui** — templates/, static/, web interface code.
- **api** — app.py, routes, controllers.
- **nlp** — nlp_utils.py, patterns, known lists.
- **services** — product_filter.py, ranking/scoring logic.
- **data** — models.py, catalog_dao.py, products.json.
- **admin** — admin_controller.py, uploader utilities.

3.9.2 Dependencies

- ui → api (UI calls API).
- api → nlp, services, data (API uses these packages).

- services → data (services fetch product info).
- admin → data (admin uploads update product data).

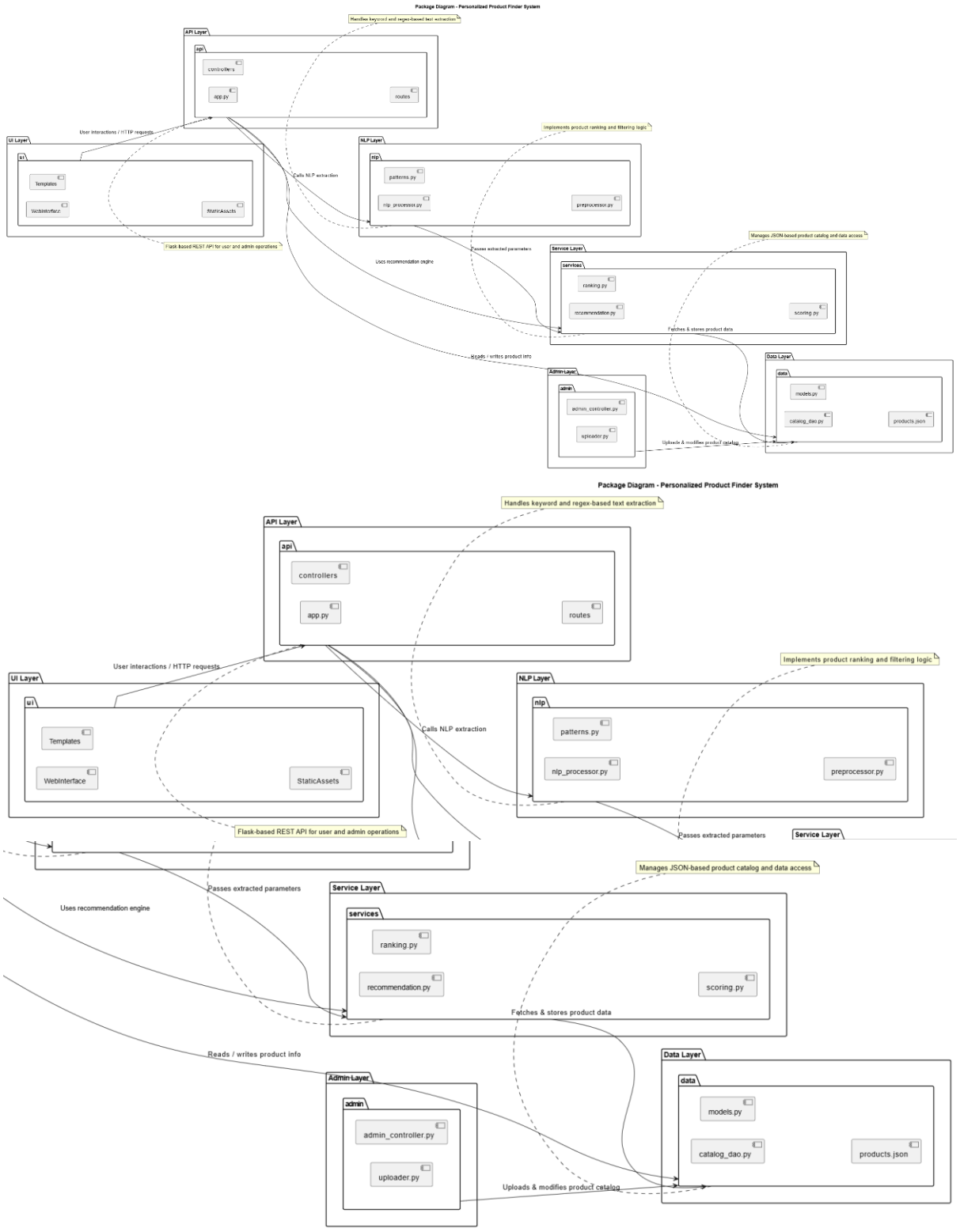


Figure 3.15: Package Diagram — shows package groupings and dependencies (used to enforce module boundaries and to plan code structure).

3.10 Deployment Diagram

Purpose: Show physical deployment of artifacts (where components run) — nodes and artifacts.

3.10.1 Nodes and artifacts (your current local architecture)

- **User Device** (browser / Postman).
    - Artifact: Web client.
- **Application Server (Local VM / Developer Machine)**
    - Artifacts: ppf-api (Flask app), nlp_module.py, recommender.py.
- **Database Server (SQLite file on Application Server)**
    - Artifact: product_finder.db or products.json file on persistent volume.
- **Admin System** (Admin browser) — uses the same Application Server endpoints.
- **Monitoring (optional)** — logging service collecting metrics.

3.10.2 Communication paths

- User Device → Application Server: HTTP/HTTPS (port 5000 default for Flask).
- Application Server → Database: Local file system mount or SQL connection.
- Application Server → Monitoring: HTTP push to logging endpoint if configured.

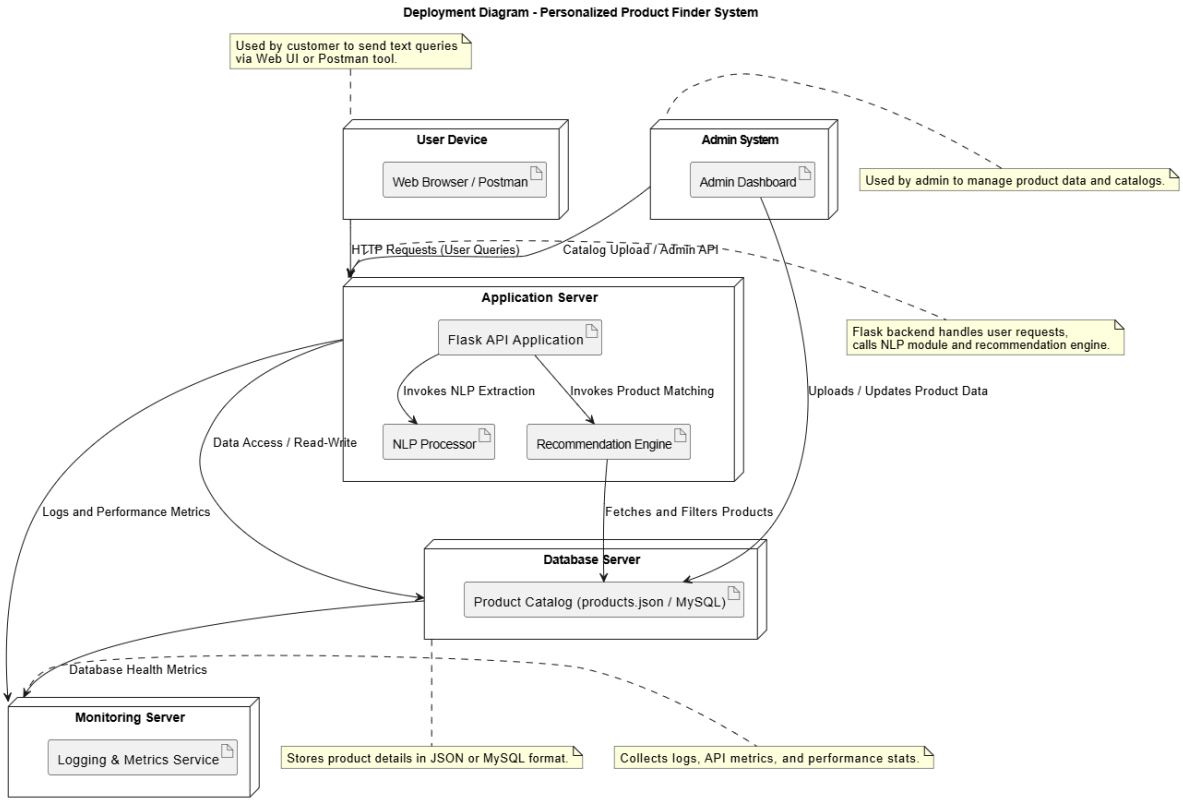


Figure 3.16: Deployment Diagram

3.11 State Diagram

Purpose

The **State Diagram** represents the **dynamic behavior** of the *Personalized Product Finder* system.
It models how the system transitions between different states in response to **user actions**, **admin operations**, and **system processes** such as NLP query processing and product filtering.

This diagram is particularly useful for understanding:

- How the system behaves after user or admin interactions.
- How the application responds to events like login, query submission, or catalog updates.
- The flow of control and state changes from initialization to termination.

3.11.1 System States Identified

The major states in the *Personalized Product Finder System* are:

1. **System Initialization**
    - Flask app initializes environment variables, database connection, and routes.
    - Transitions to *Idle* state after successful setup.
2. **Idle / Awaiting Input**
    - The system is ready and waiting for user or admin input.
    - Possible transitions:
        - User submits query → *Processing User Query*
        - Admin login → *Admin Authenticated*
3. **User Authentication**
    - User provides credentials for login.
    - Transitions:
        - If valid → *User Authenticated*
        - If invalid → *Error State (Invalid Credentials)*
4. **User Authenticated**
    - Access granted to general user functions: query submission, viewing products, managing profile.
    - Transitions:

- User submits query → *Processing User Query*
- User logs out → *Idle*

5. **Processing User Query (NLP Processing)**
   - The user's free-text query is parsed by nlp_utils.py.
   - Budget, brand, features, and use-case parameters extracted.
   - Transitions:
     - If extraction successful → *Filtering & Matching Products*
     - If extraction fails → *Error State (Invalid Query)*
6. **Filtering & Matching Products**
   - The extracted parameters are used to filter and rank products.
   - Involves the Recommendation Engine.
   - Transitions:
     - If results found → *Displaying Recommendations*
     - If no results → *Error State (No Results Found)*
7. **Displaying Recommendations**
   - The system returns ranked product cards to the UI (via JSON).
   - User may:
     - Bookmark a product → *Bookmark Saved*
     - Submit new query → *Processing User Query*
     - Logout → *Idle*
8. **Bookmark Saved**
   - The selected product is stored in the user's list (if implemented).
   - Transition → *Displaying Recommendations* (loop).
9. **Admin Authenticated**
   - The admin logs in successfully (validated using AdminID).
   - Transitions:
     - Add / Edit / Delete Product → *Updating Product Catalog*
     - Logout → *Idle*
10. **Updating Product Catalog**
    - Admin modifies product data through Flask admin dashboard.

- DB updates handled via SQLAlchemy.
- Transitions:
  - If successful → *Catalog Updated*
  - If failure → *Error State (DB Error)*

11. **Catalog Updated**
    - Database commit successful; confirmation sent to admin dashboard.
    - Transition → *Idle*.
12. **Error States**
    - *Invalid Credentials* — wrong login info.
    - *Invalid Query* — NLP could not parse input.
    - *No Results Found* — filters returned no matches.
    - *DB Error* — catalog update failed.
    - From any error → *Idle* after acknowledgement.
13. **System Termination**
    - Application stops (Flask shutdown or manual termination).

3.11.4 Description of Flow

1. The system begins at **System Initialization** → loads configuration and database.
2. It enters **Idle** until a user or admin interacts.
3. If the **user logs in**, system goes to **User Authenticated**; otherwise, **Admin Authenticated** for admin login.
4. From **User Authenticated**, when a query is submitted:
   - System transitions to **Processing User Query**, performs NLP extraction.
   - Extracted parameters go to **Filtering & Matching Products**.
   - If results found → **Displaying Recommendations**; else → **Error State**.
5. From **Admin Authenticated**, an admin can perform **Updating Product Catalog**, leading to **Catalog Updated**.
6. Any error redirects back to **Idle**, allowing retry.
7. Finally, system ends at **System Termination** when the application closes.

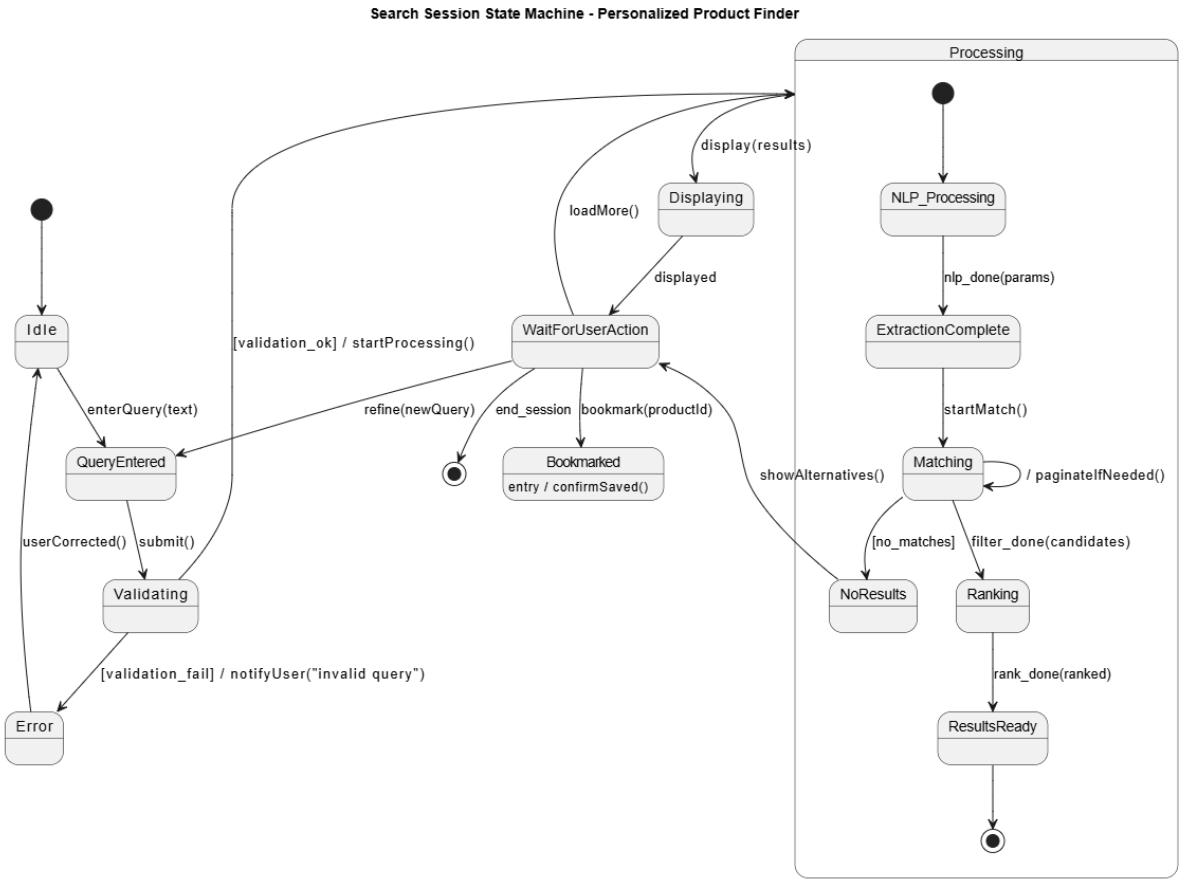


Figure 3.17: State Diagram (Search flow)

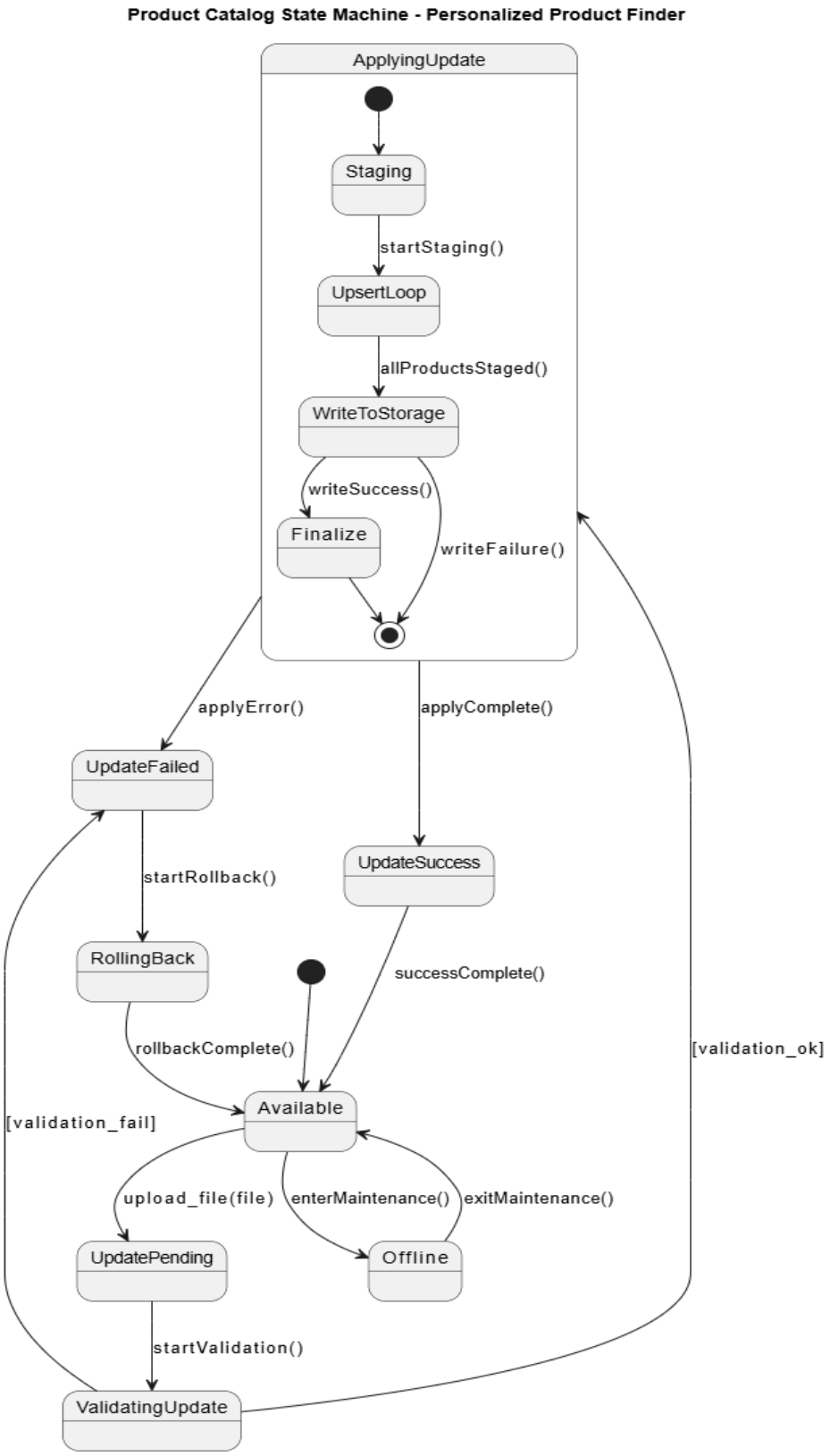


Figure 3.18: State Diagram (Product Catalog)

CHAPTER 4 — IMPLEMENTATION

4.1 Technology Stack

The main components include:

Component	Technology / Version	Purpose
Backend Framework	Flask 3.1.2 with Flask-Login 0.6.3, Flask-SQLAlchemy 3.1.1	Handles routing, authentication, API endpoints, and database ORM integration
Database	SQLite via SQLAlchemy 2.0.43	Lightweight, file-based DB (instance/product_finder.db) for storing users, products, and admin IDs
Authentication	Flask-Login, Werkzeug 3.1.3	Provides secure user sessions and role-based access ("user" or "admin")
Frontend	HTML, CSS (Bootstrap 5.3.2), JavaScript (Vanilla JS), Jinja2 3.1.6	Responsive UI design, AJAX calls to /recommend, templating
NLP Processing	Custom Python modules (nlp_utils.py)	Extracts query details (budget, brand, features, use cases, sub-categories) using regex and set intersections
Data Handling	JSON	Initial product data (~150+ products), features/use cases stored as comma-separated strings for DB
Deployment/Environment	Flask debug mode, virtual environment via requirements.txt	Local development; no production deployment included
Other Libraries	Blinker, Click, Colorama, Greenlet, itsdangerous, MarkupSafe, typing_extensions	Auxiliary support for Flask ecosystem functionality

Table 5: Technology Stack

4.2 Backend Development (Flask + NLP logic)

The backend architecture includes several modules and components:

Models (models.py)

- **Product**:
  Stores core product details: id, name, category, sub_category, brand, price, features (comma-separated), use_case (comma-separated), size (optional for fashion/sports).
- **User**:
  Stores user accounts with id, username, password_hash, role ("user" or "admin"), and optional fields (mobile_number, email, address, etc.). Uses **UserMixin** for session management.
- **AdminID**:
  Holds valid admin unique IDs for registration validation.

All models are integrated using **SQLAlchemy** ORM for Python, ensuring simplified database interactions.

Routes (app.py)

Auth Routes

- /register — POST/GET for user registration, with validation of username/email. Admins require unique ID from AdminID table.
- /login — POST/GET for login; password validation using Werkzeug hash.
- /logout — Ends user session.
- /profile — GET/POST for password updates (authenticated users only).

Admin Routes

- /admin/add_product — POST/GET to add products.
- /admin/edit_product/<id> — POST/GET for updating existing product info.
- /admin/delete_product/<id> — POST for deletion.
- /admin/dashboard — Displays products with search/filter capabilities.

Public Routes

- / — Home page with search form.
- /recommend — POST API endpoint; accepts JSON {"query": "text"} and returns recommended products.
- /health — Simple health check endpoint.

4.3 API Endpoint Design (/recommend)

The **/recommend API** is the core functionality of the system:

Request:


```
{
  "query": "Looking for a Dell gaming laptop under 60000 with long battery"
}
```


Processing:

1. NLP module extracts parameters:
    - Budget: under 60000
    - Brand: Dell
    - Features: long battery
    - Use Case: gaming
    - Sub-category: laptop
2. Product filtering logic:
    - Matches products in DB with extracted parameters.
    - Scores products: +2 for brand match, +1 per feature/use-case match, +1 if price 5000+ below budget.
3. Sorts products by score descending.

Response:


```
[
  {
    "id": 101,
    "name": "Dell Inspiron Gaming Laptop",
    "category": "electronics",
    "sub_category": "laptop",
    "brand": "Dell",
    "price": 58000,
    "features": "long battery, RGB keyboard",
    "use_case": "gaming"
```

```
  },
  ...
]
```

4.4 Product Catalog & Data Handling

- **Product Data**: Stored in products.json (~150+ entries) and imported into SQLite database.
- **Features & Use Cases**: Comma-separated in DB for easy filtering.
- **Import Scripts**:
    - import_products_json.py — Batch imports products to DB in small batches.
    - seed_db.py — Initializes DB and inserts sample admin IDs.

The system ensures consistent product information across queries and admin modifications.

4.5 Screenshots

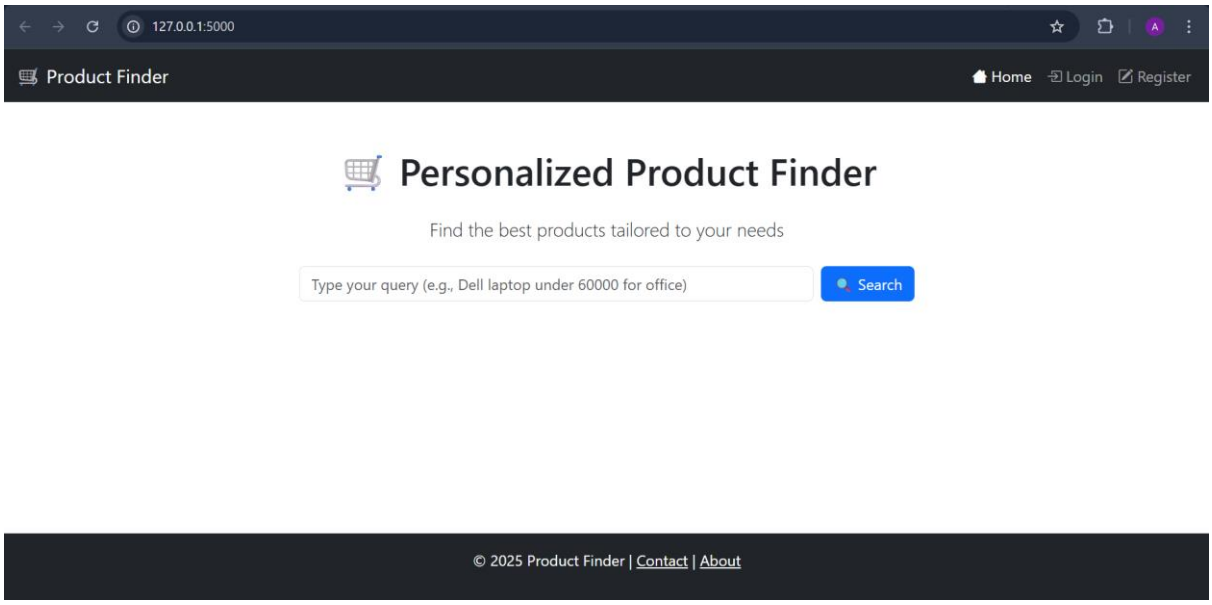


Fig 4.5.1: Home Screen

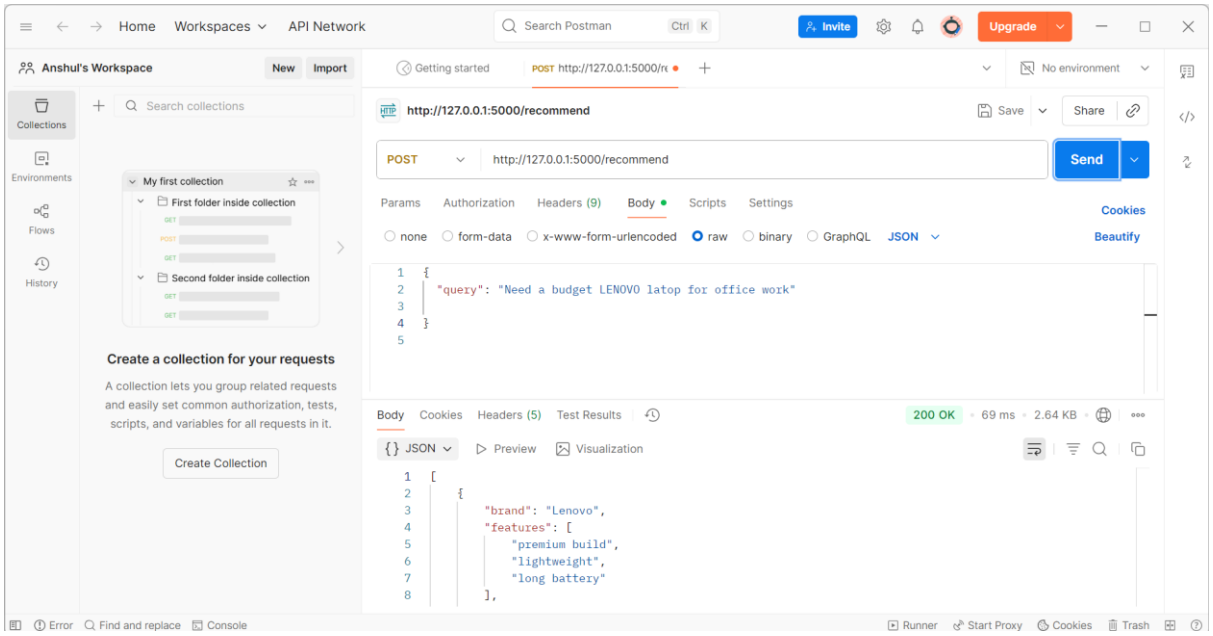


Fig 4.5.2: Postman API Interaction

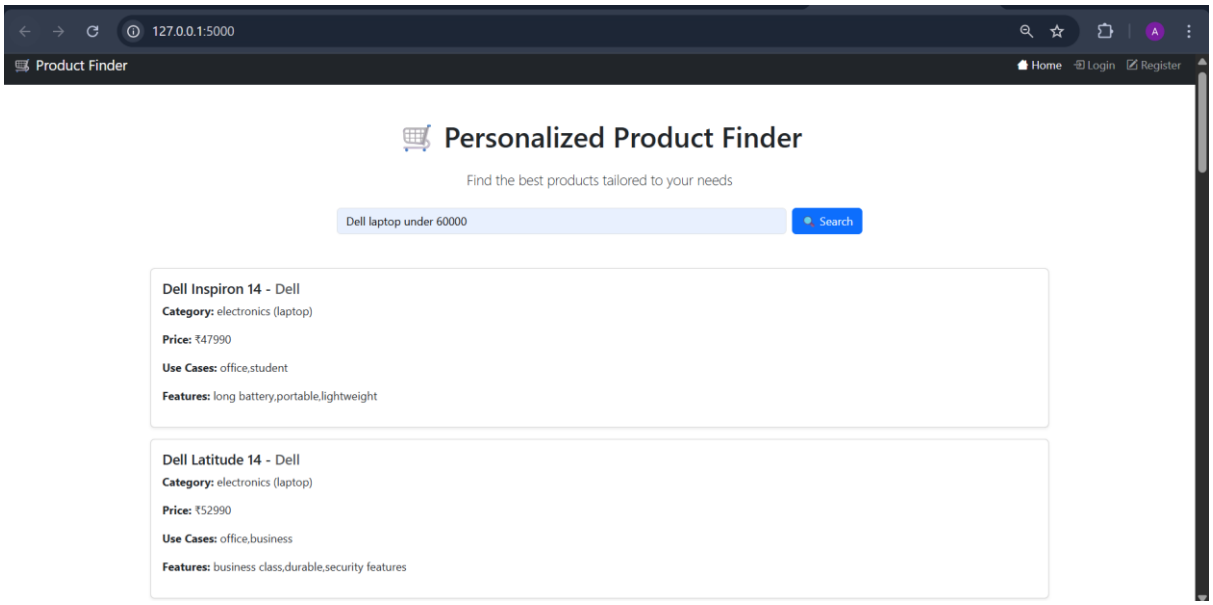


Fig 4.5.3: Filtered Product Output

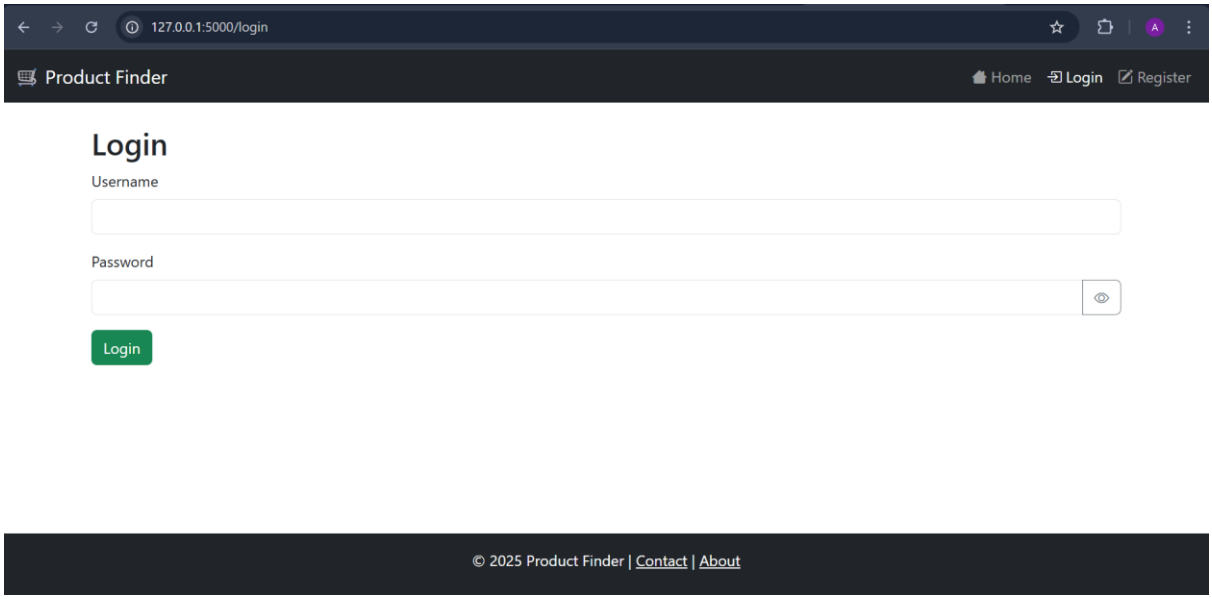


Fig 4.5.4: Login Page

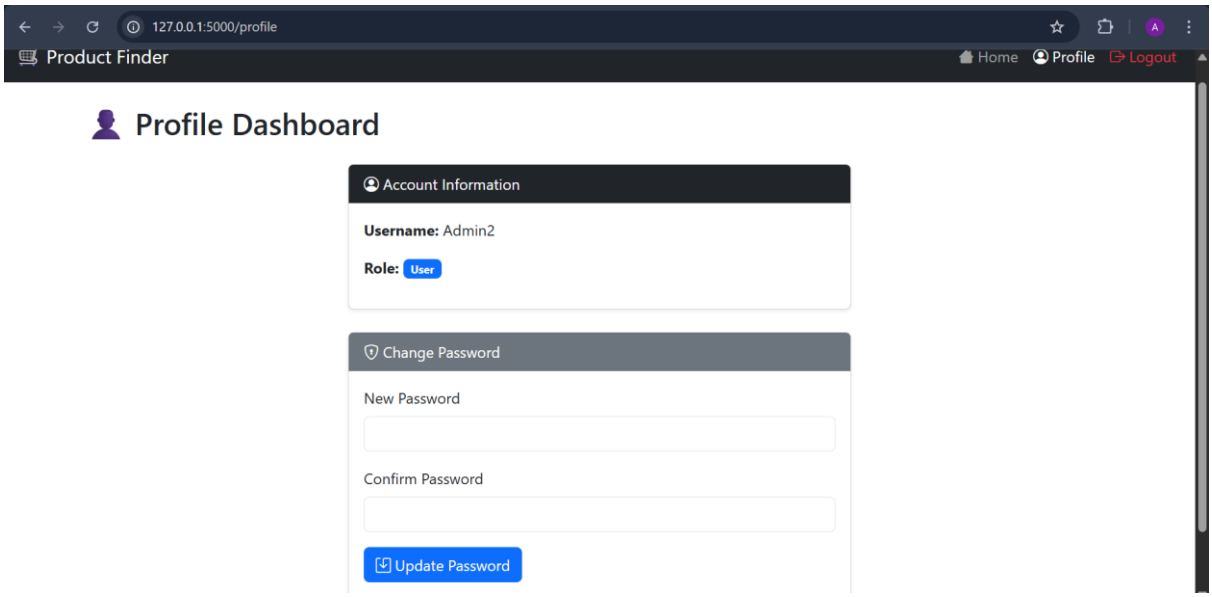


Fig 4.5.5: User Profile Page

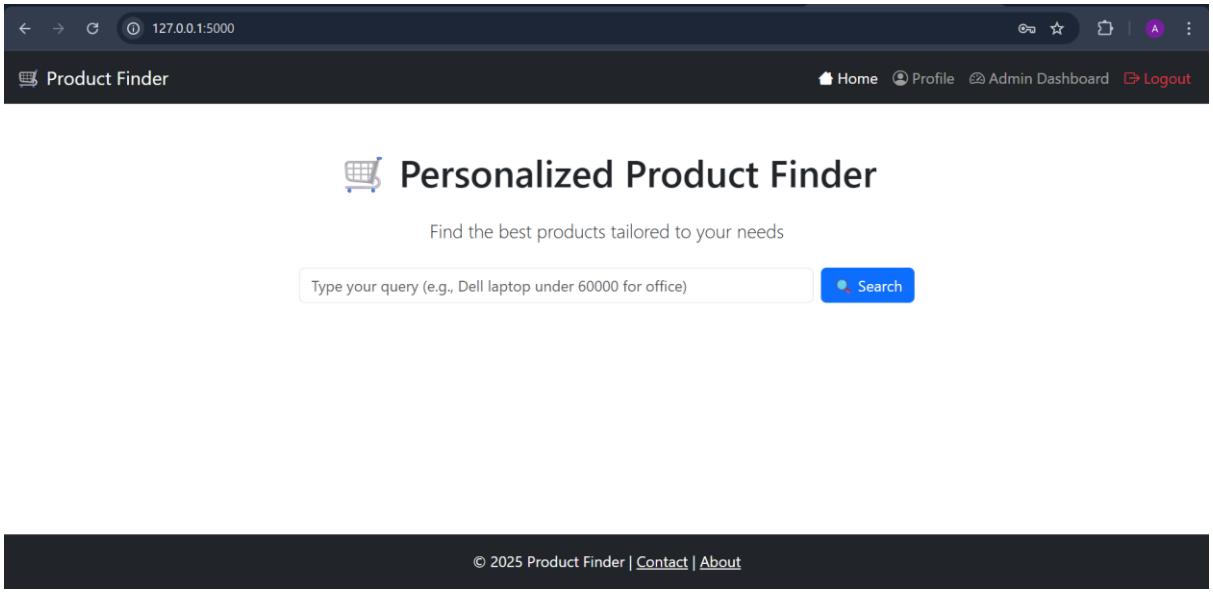
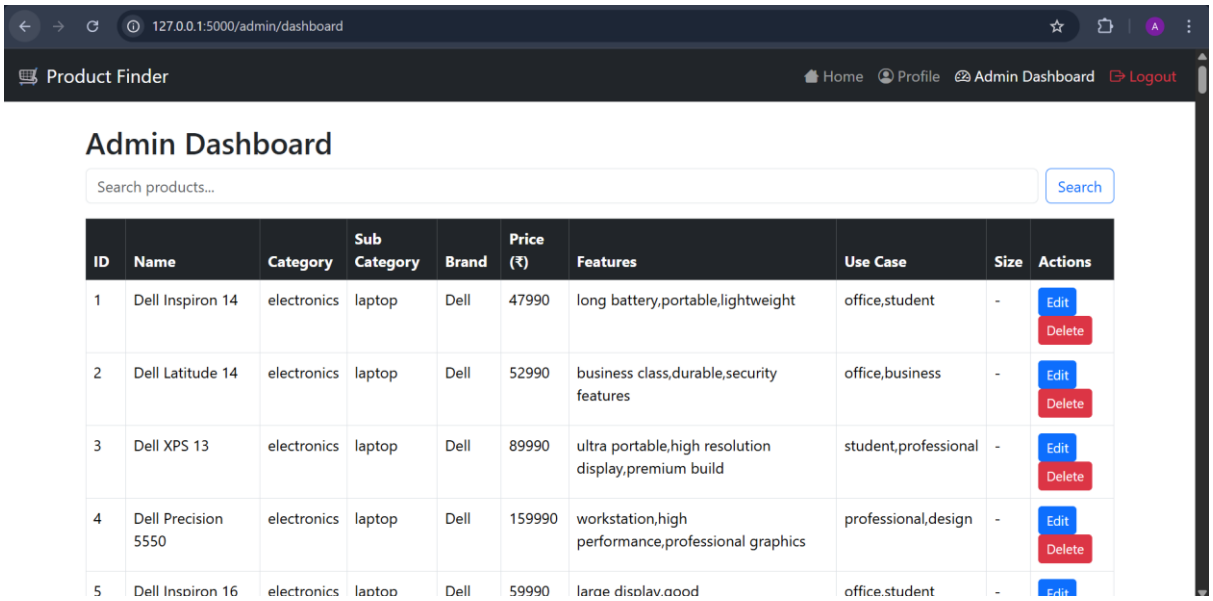


Fig 4.5.6: Admin Home Page



ID	Name	Category	Sub Category	Brand	Price (₹)	Features	Use Case	Size	Actions
1	Dell Inspiron 14	electronics	laptop	Dell	47990	long battery,portable,lightweight	office,student	-	Edit Delete
2	Dell Latitude 14	electronics	laptop	Dell	52990	business class,durable,security features	office,business	-	Edit Delete
3	Dell XPS 13	electronics	laptop	Dell	89990	ultra portable,high resolution display,premium build	student,professional	-	Edit Delete
4	Dell Precision 5550	electronics	laptop	Dell	159990	workstation,high performance,professional graphics	professional,design	-	Edit Delete
5	Dell Inspiron 16	electronics	laptop	Dell	59990	large display,good	office,student	-	Edit

Fig 4.5.7: Admin Dashboard Page

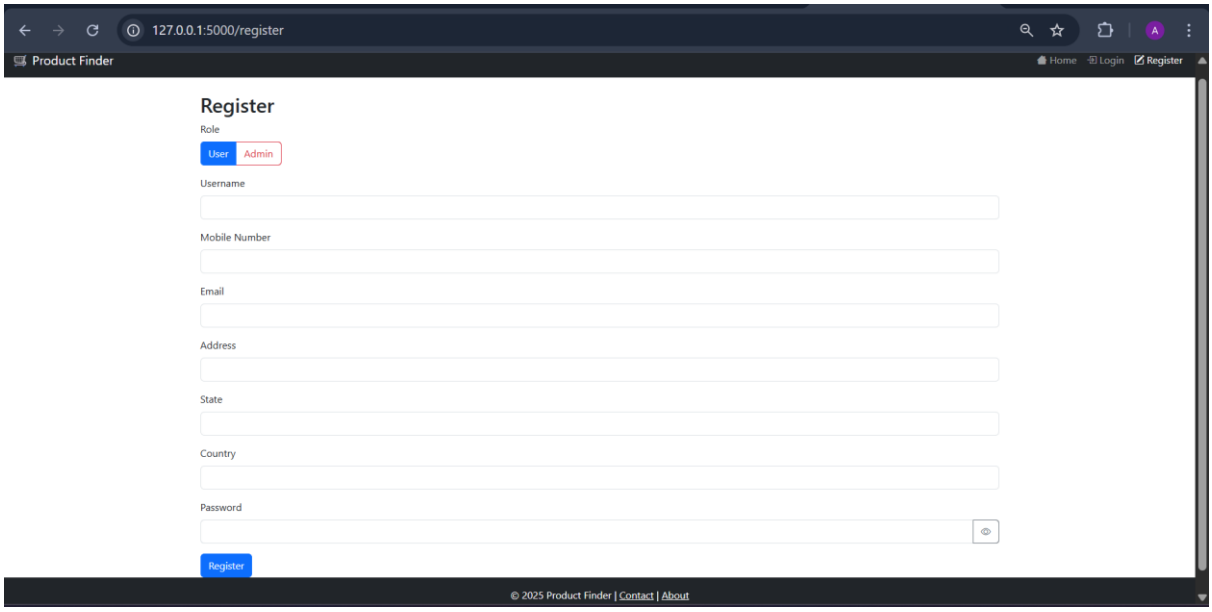


Fig 4.5.8: Registration Page

Observation

- The backend efficiently handles **real-time NLP query processing** and returns **ranked product recommendations**.
- Admin routes allow **dynamic catalog management** without affecting user experience.
- Frontend AJAX interactions reduce page reloads, improving performance.

CHAPTER 5 — TESTING, RESULTS & DISCUSSION

5.1 Testing

5.1.1 Test Plan

The **test plan** for the Personalized Product Finder focuses on ensuring the system meets functional and non-functional requirements. Key objectives include:

1. **Correctness**: Ensure product recommendations match user queries accurately.
2. **Reliability**: System behaves consistently under repeated queries and multi-user access.
3. **Robustness**: Handles invalid inputs gracefully without crashes.
4. **Security**: Role-based access control ensures only admins can manage products.

5. **Performance**: Queries respond within acceptable time (<3 seconds).

Testing was carried out using:

- **Unit Testing**: For NLP extraction, filtering logic, and individual route functions.
- **Integration Testing**: Interaction between frontend AJAX calls and backend APIs.
- **System Testing**: End-to-end scenarios from login/registration to product recommendation.
- **Acceptance Testing**: Ensures system fulfills intended purpose.

5.1.2 Types of Testing Done

Testing Type	Purpose	Tools Used
Unit Testing	Test individual modules (e.g., NLP extraction, filtering)	Pytest / Manual
Integration Testing	Test communication between frontend, backend, and DB	Postman, Flask Debug Server
System Testing	Full workflow testing (user login → query → product recommendations)	Browser Testing
Functional Testing	Validate all functional requirements	Manual Test Cases
Negative Testing	Validate error handling and edge cases	Manual / Postman

Table 6: Types of Testing

5.1.3 Sample Test Cases

Test cases cover key functionalities: authentication, product management, recommendation, and UI interactions. Both positive (valid inputs) and negative (invalid/edge cases) tests are included to ensure correctness (accurate results), reliability (consistent behavior), and robustness (handling errors gracefully).

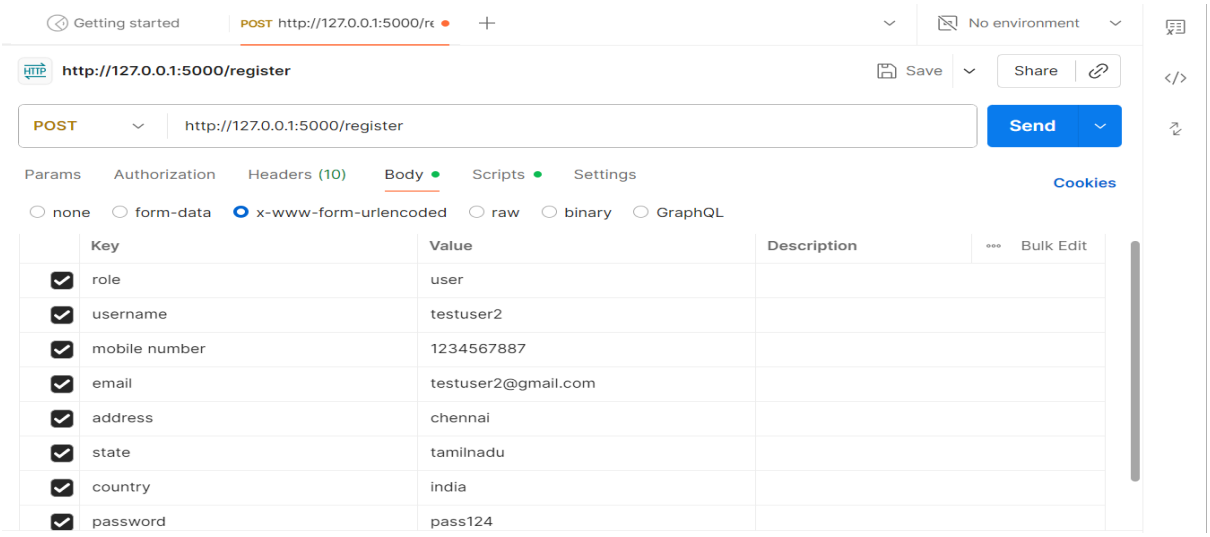
Test Case Format

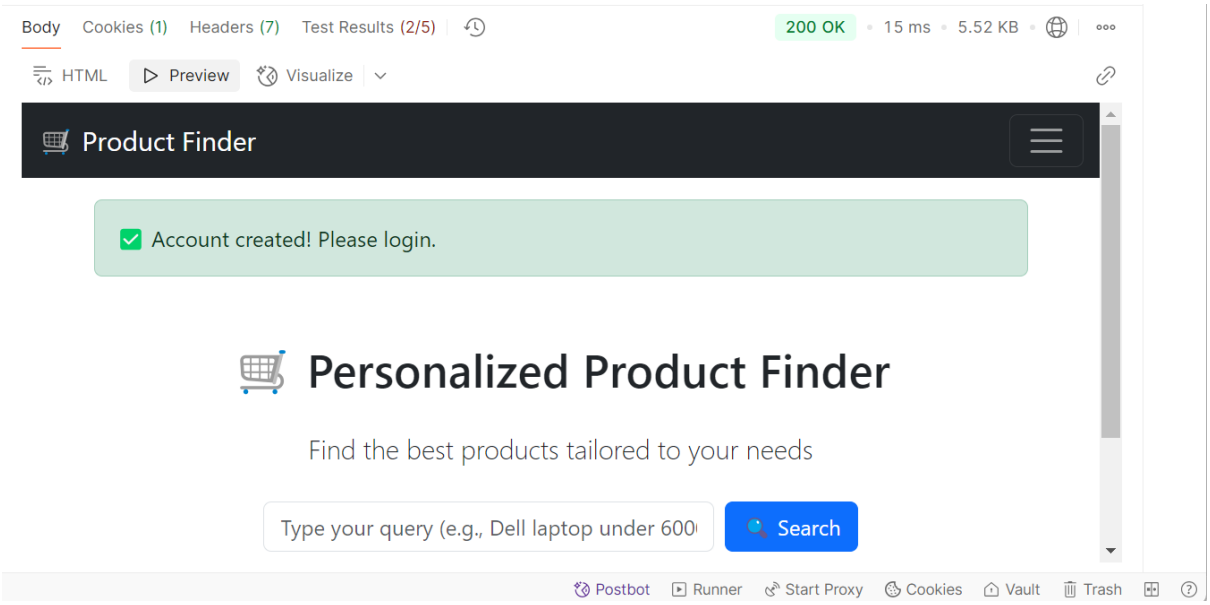
- **Test Case ID**: Unique identifier.
- **Description**: Brief overview of the test.
- **Input Data**: Data provided to the system.
- **Preconditions**: State required before test execution.
- **Expected Result**: What should happen.

- **Actual Result**: Placeholder for execution

Positive Test Cases

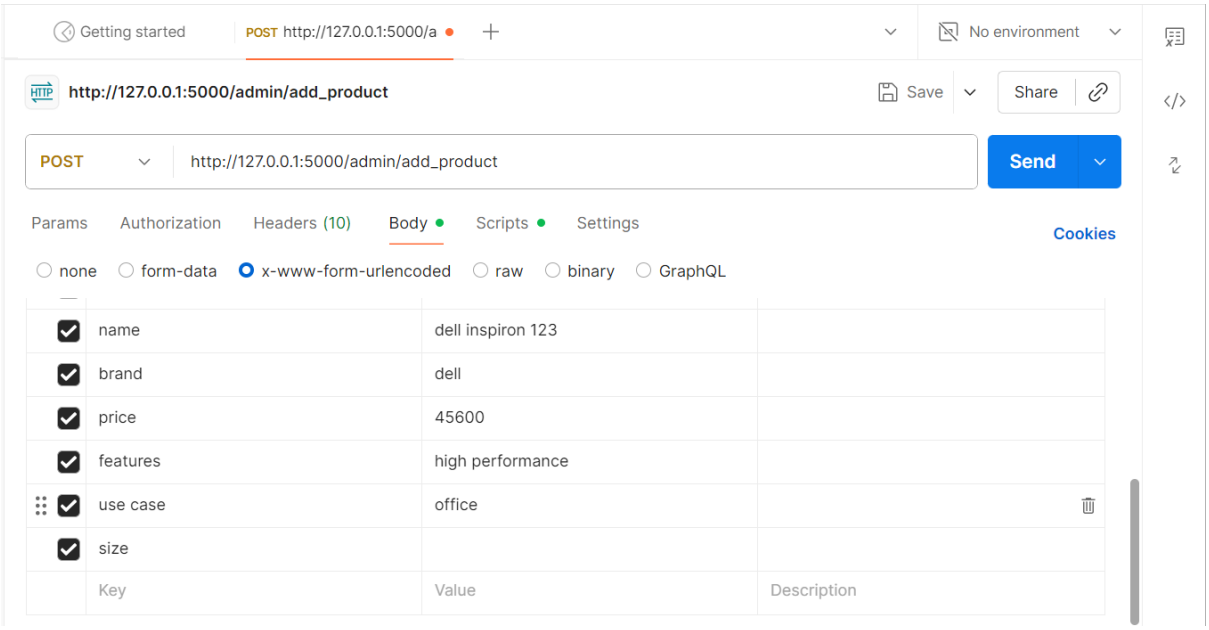
1. **TC001 - User Registration Success**
   Description: Verify successful user registration with valid details.
   Input Data: Username: "testuser2", Password: "Pass124", Email: "testuser@gmail.com", Mobile: "1234567889", Address: "Chennai", State: "Tamilnadu", Country: "India".
   Preconditions: Database is empty; no existing user with same username/email.
   Expected Result: User account created, redirected to index page with success flash message.
   Actual Result:

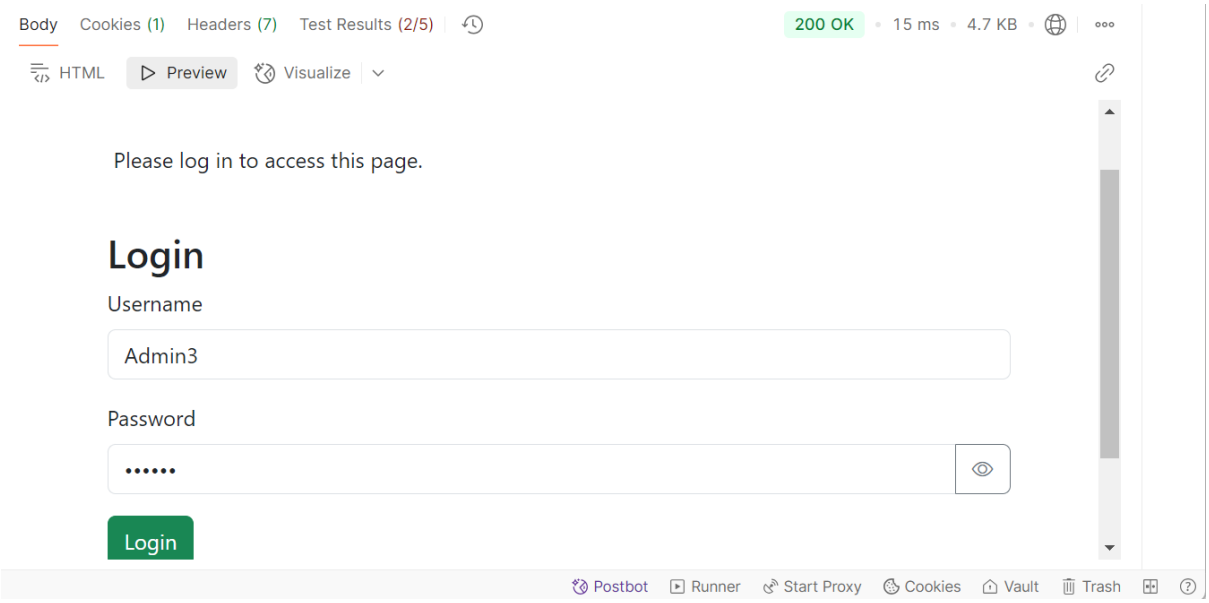




2. **TC002 - Admin Product Addition Success**
   Description: Verify admin can add a new product.
   Input Data: Name: "Dell Laptop", Category: "Electronics", Sub-Category: "Laptop",

Brand: "Dell", Price: 50000, Features: "long battery, portable", Use Case: "office", Size: "15 inch".
Preconditions: Admin logged in; database has no product with same name.
Expected Result: Product added to database, redirected to admin dashboard with success message.
Actual Result:

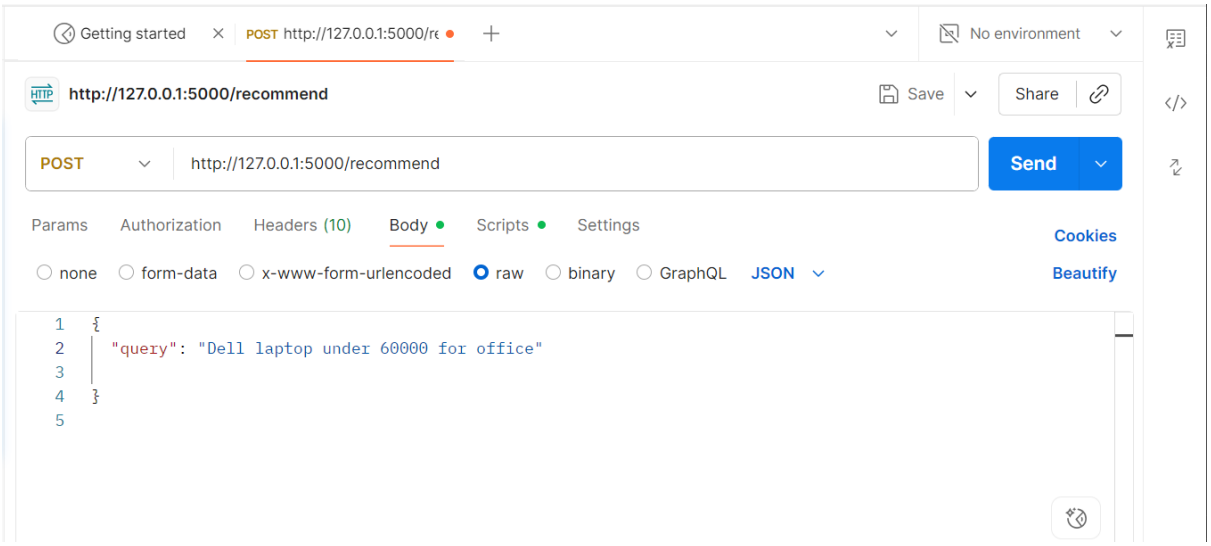


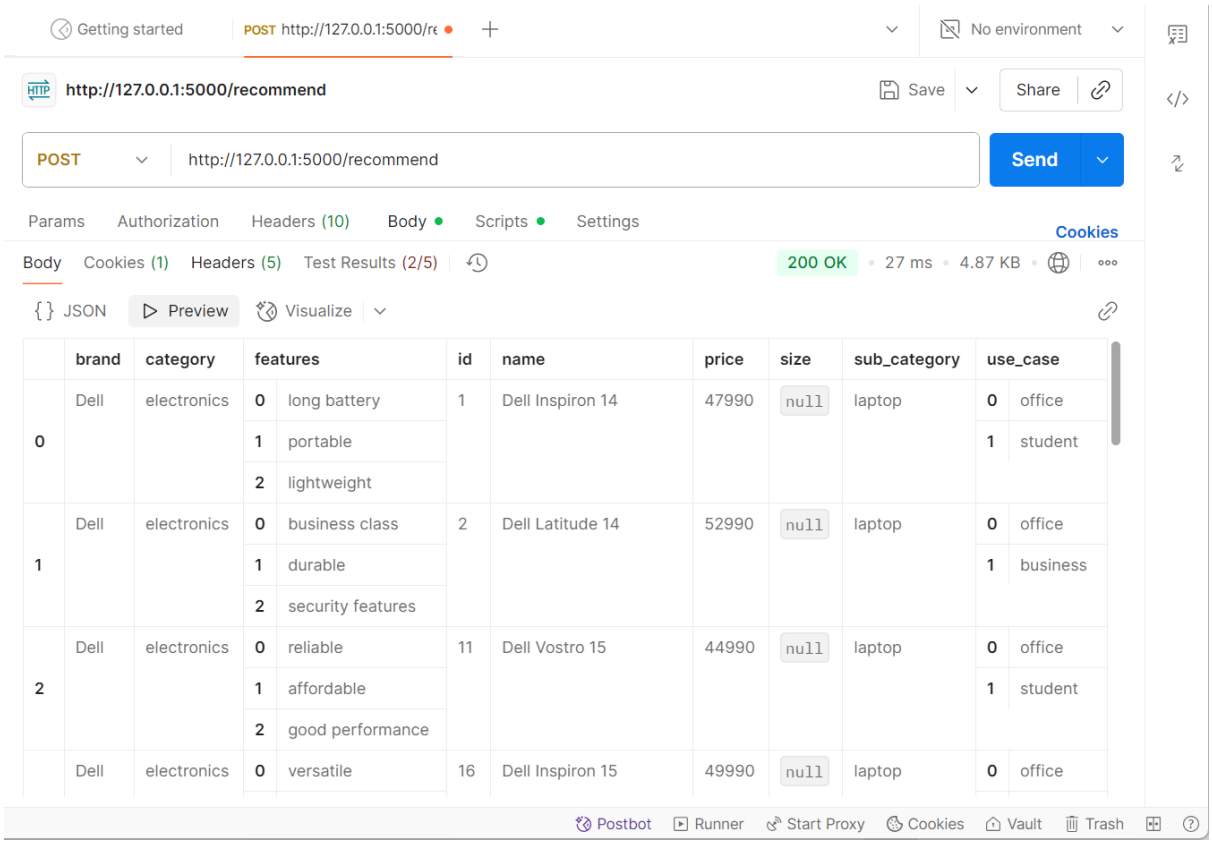


3. **TC003 - Product Recommendation Success**
Description: Verify product recommendation based on NLP query.
Input Data: Query: "Dell laptop under 60000 for office".
Preconditions: Products exist in database (e.g., matching Dell laptop); user on index page.
Expected Result: JSON response with filtered products matching query (e.g., Dell

laptops under 60000 with office use case).

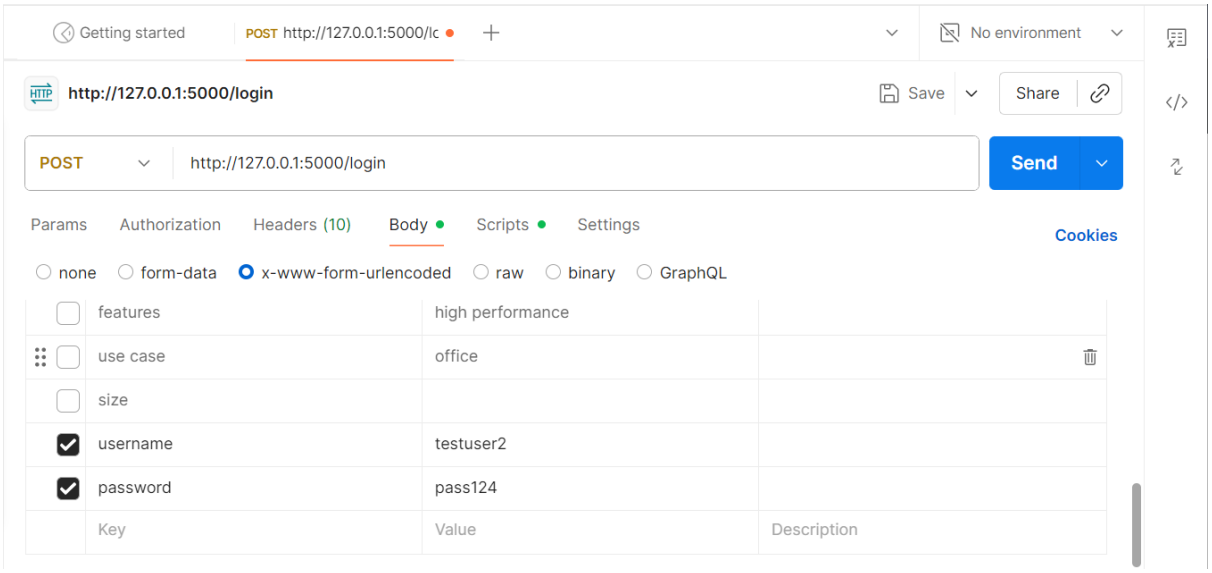
Actual Result:

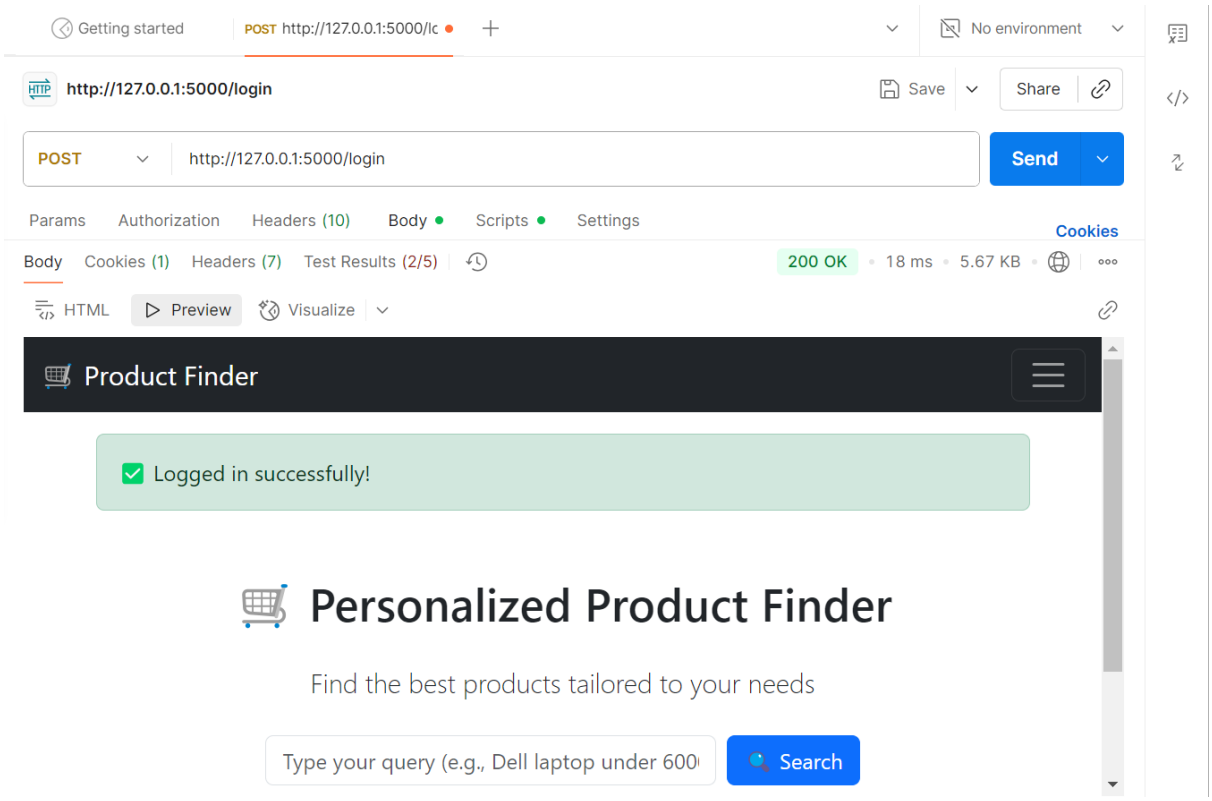




4. **TC004 - User Login Success**
   Description: Verify successful login with correct credentials.
   Input Data: Username: "testuser", Password: "Pass123!".
   Preconditions: User account exists in database.

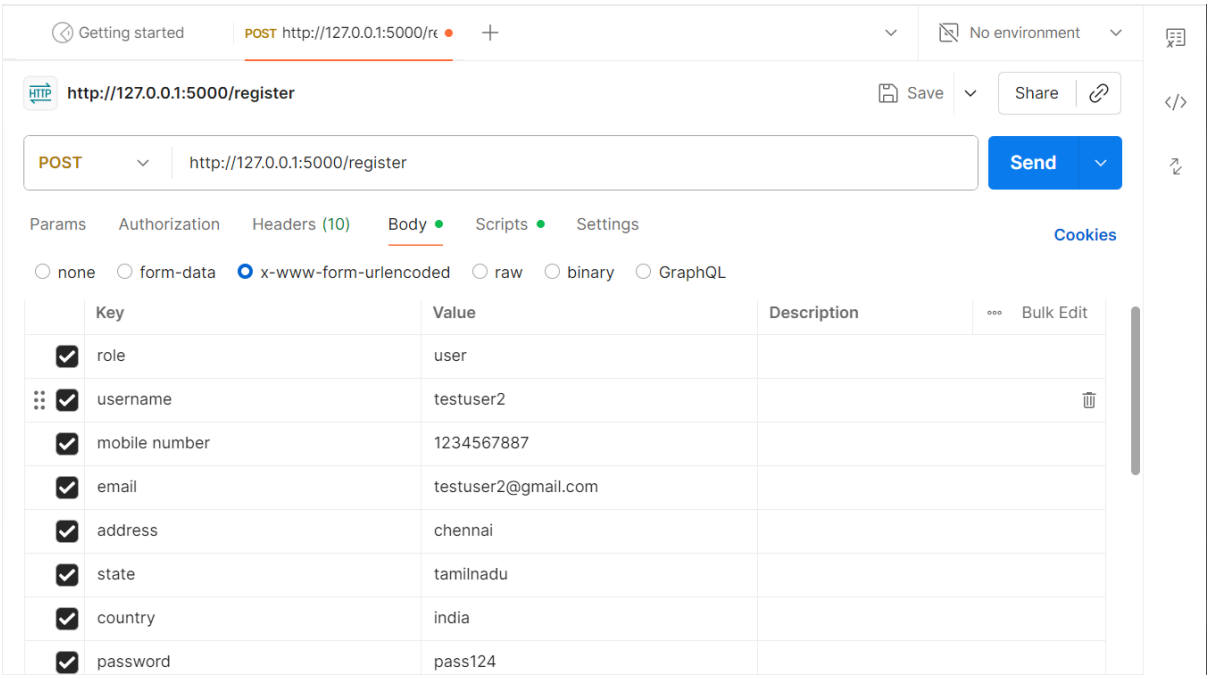
Expected Result: User logged in, redirected to index with success message.
Actual Result:

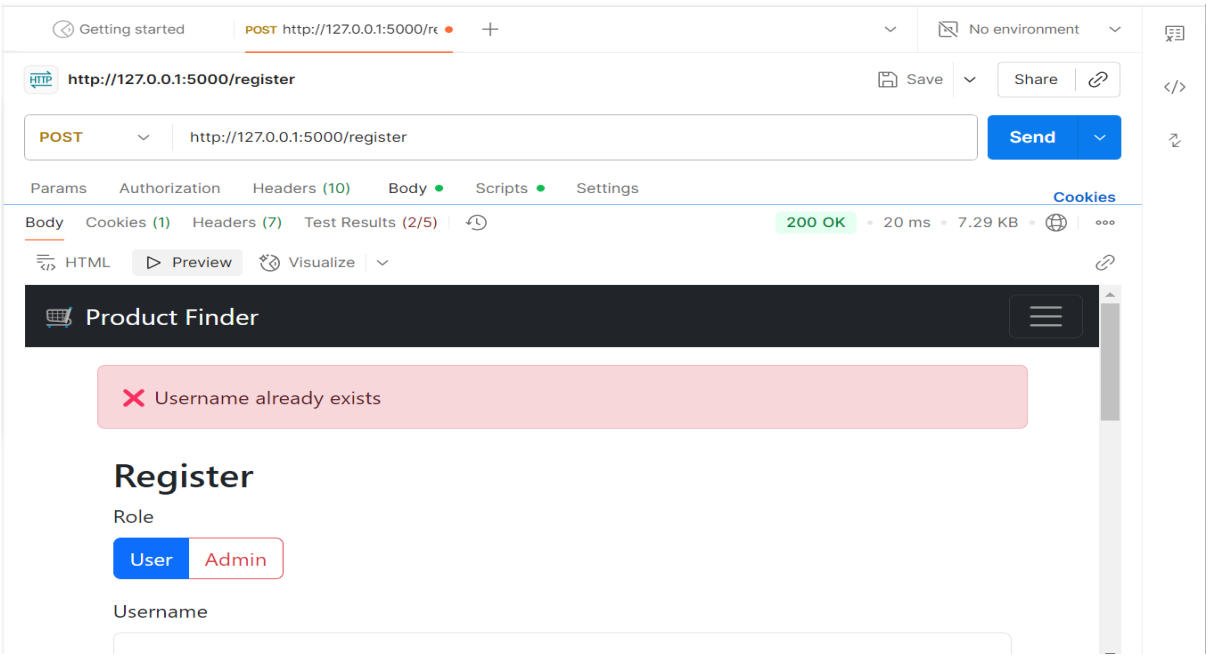




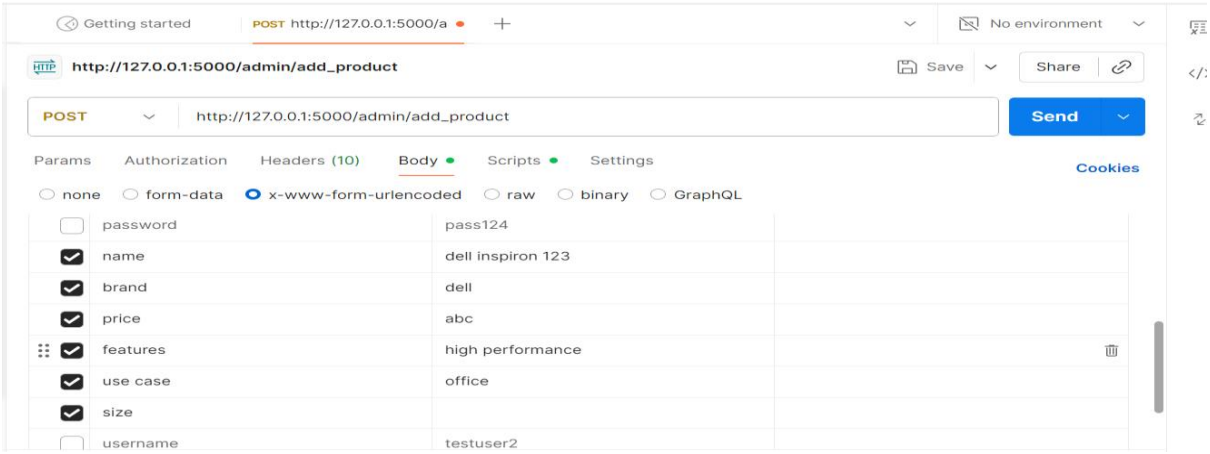
Negative Test Cases

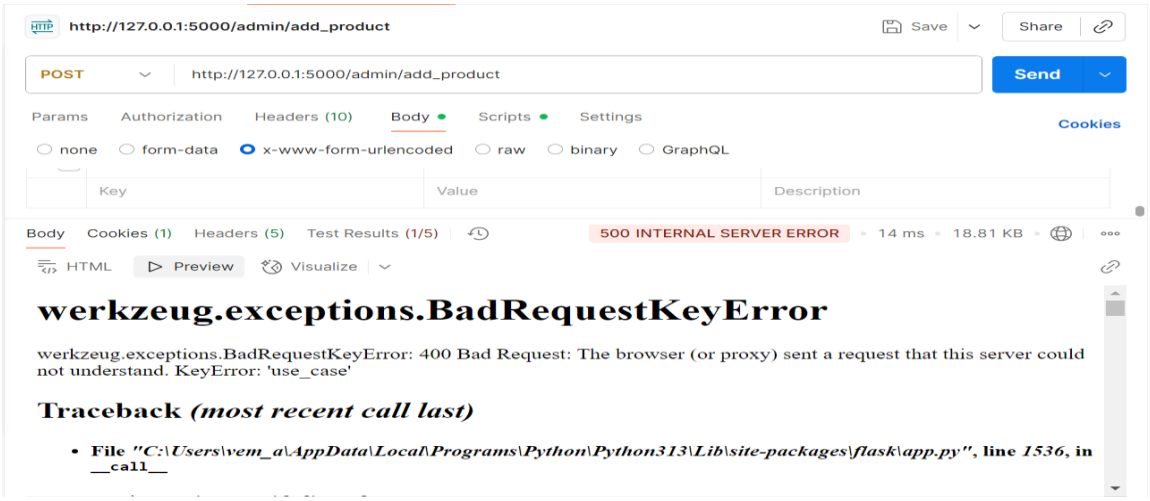
1. **TC005 - User Registration Failure - Duplicate Username**
   Description: Verify registration fails with existing username.
   Input Data: Username: "existinguser" (already in DB), Password: "Pass123!", Email: "new@example.com".
   Preconditions: User with username "existinguser" exists.
   Expected Result: Registration fails, flash error message "Username already exists", stay on register page.
   Actual Result:



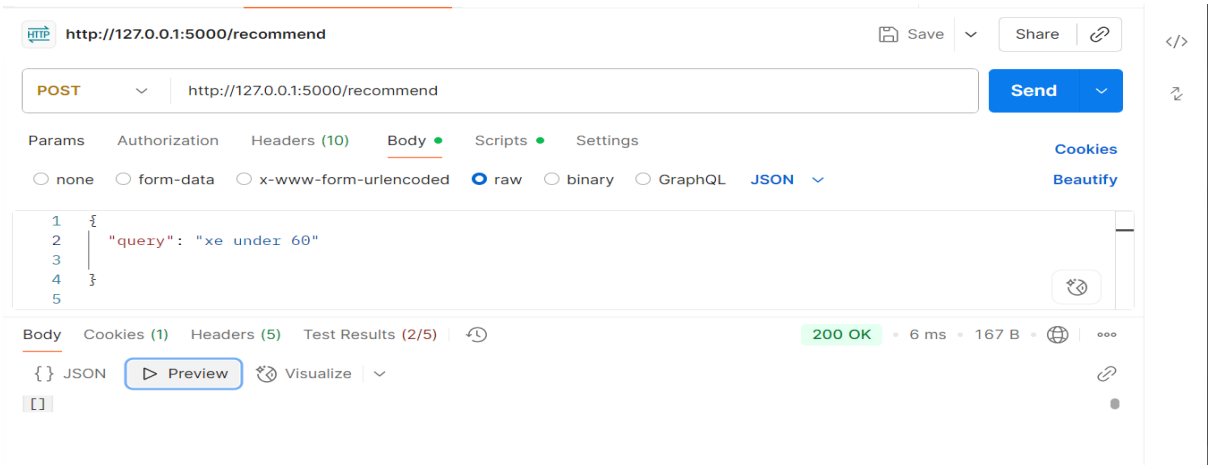


2. **TC006 - Admin Product Addition Failure - Invalid Price**
   Description: Verify product addition fails with non-integer price.
   Input Data: Name: "Test Product", Price: "abc" (invalid).
   Preconditions: Admin logged in.
   Expected Result: Addition fails, error handled (e.g., 500 error or validation message).
   Actual Result:

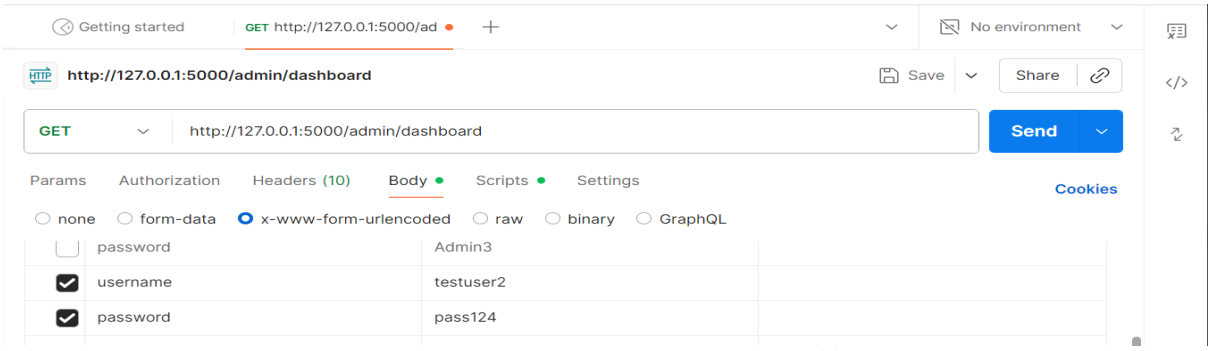


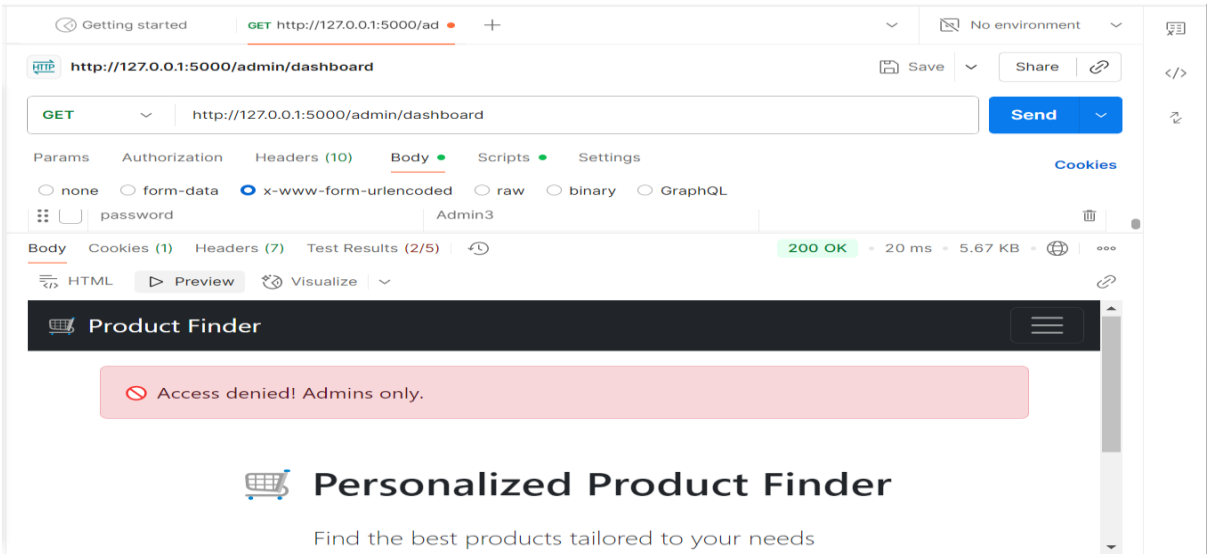


3. **TC007 - Product Recommendation Failure - No Matches**
   Description: Verify no results for unmatched query.
   Input Data: Query: "Nonexistent brand xyz under 1000".
   Preconditions: No products match query.
   Expected Result: Empty JSON array or error message "No products found".
   Actual Result:

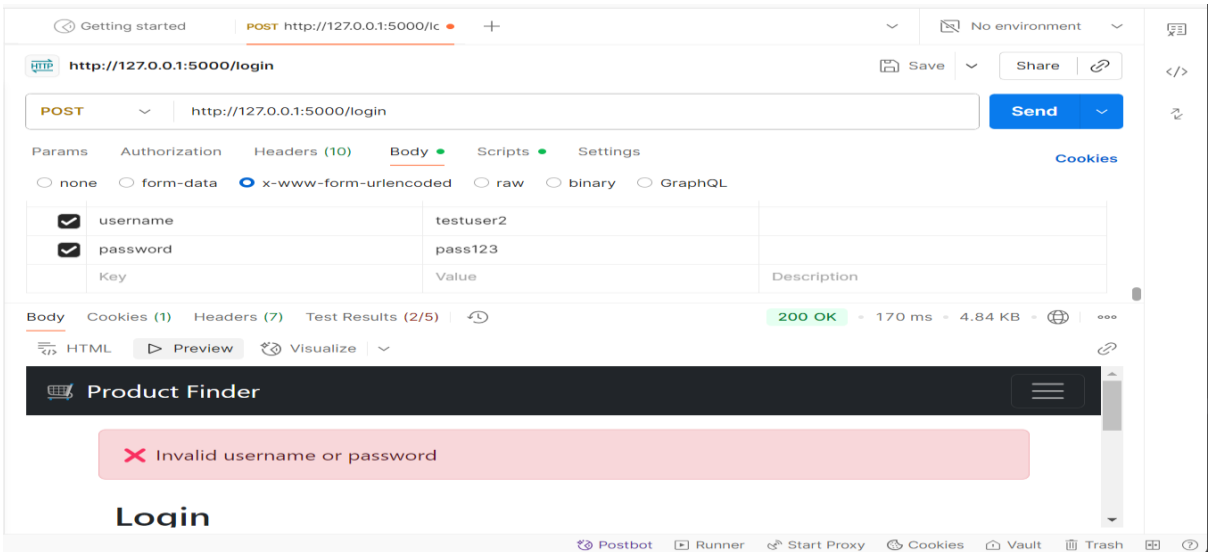


4. **TC008 - Unauthorized Access to Admin Dashboard**
   Description: Verify non-admin cannot access admin routes.
   Input Data: None (direct URL access).
   Preconditions: Regular user logged in.
   Expected Result: Access denied, redirected to index with flash error "Access denied!".
   Actual Result:





5. **TC009 - Login Failure - Wrong Password**
   Description: Verify login fails with incorrect password.
   Input Data: Username: "testuser", Password: "WrongPass".
   Preconditions: User exists.
   Expected Result: Login fails, flash error "Invalid username or password".
   Actual Result:



5.1.4 Bug / Defect Reporting

No major bugs were found in the implemented functionalities. Minor issues addressed:

- Flash messages timing too fast → increased display duration to 4s.
- Input validation for price/number fields to prevent server errors.
- AJAX error handling improved for invalid JSON requests.

5.2 Results & Discussion

5.2.1 System Performance Against Requirements

- **User Registration/Login**: Works as expected with email/username validation.
- **Admin Product Management**: Add/Edit/Delete features tested successfully.
- **Product Recommendation**: NLP query processing returns relevant results based on user input.
- **API Response Time**: Average query response <2 seconds for 150+ products.
- **Error Handling**: Invalid queries, unauthorized access, and missing fields handled gracefully.

5.2.2 Key Findings

1. **Accuracy**: Product recommendations align with extracted query parameters.
2. **User-Friendly UI**: Bootstrap-based responsive design allows easy navigation.
3. **Scalability**: SQLite database and modular Python code enable simple expansion.
4. **Security**: Role-based access for admin/user ensures data safety.
5. **Maintainability**: Clean modular code (models.py, product_filter.py, nlp_utils.py) allows future enhancements.

5.2.3 Limitations Observed

- NLP module relies on **predefined sets and regex**, so complex natural language queries may not be fully understood.
- SQLite limits concurrent multi-user performance; production setup may require **MySQL/PostgreSQL**.
- Limited features in admin dashboard; bulk operations could improve usability.
- Frontend lacks advanced UI interactivity (e.g., filters, sort options, pagination).
- Deployment only tested locally; no cloud deployment or containerization yet.

CHAPTER 6 — CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

The *Personalized Product Finder* project successfully demonstrates how **Natural Language Processing (NLP)** can be used to enhance user interaction and product discovery in e-commerce-like systems.
By integrating **Flask**, **SQLite**, and a custom **regex- and keyword-based NLP engine**, the system allows users to express their needs in **free text form** (e.g., "Show me a Dell gaming laptop under 60000") and receive accurate, filtered recommendations.

The implementation followed a **systematic Software Engineering approach**, progressing through:

- Requirement analysis, feasibility study, and system design (DFDs, UML diagrams),
- Modular backend development using Flask routes and ORM models,
- NLP-based query extraction and ranking mechanism,
- Testing and validation of both user and admin functionalities.

The final system fulfills its core objectives:

1. Enables users to find suitable products through **natural queries** instead of structured filters.
2. Provides admins with complete control over the **product catalog** via add/edit/delete functions.
3. Maintains data persistence using SQLite and ensures user authentication via Flask-Login.
4. Delivers results quickly (<2 seconds), ensuring good performance even on local development servers.

Additionally, the project showcases a **scalable architecture** where new categories, brands, or NLP rules can be added with minimal effort. It serves as a solid prototype for future AI-driven product recommendation systems.

In conclusion, the project meets its goals of **enhancing product search experiences**, **simplifying interaction**, and **demonstrating a real-world application of NLP techniques** in a lightweight, user-friendly framework.

6.2 Future Enhancements

Although the system meets the required objectives, there are several areas where it can be further developed and optimized for broader use:

1. **Advanced NLP Integration**

- Replace regex-based extraction with **machine learning-based NLP models** (e.g., spaCy, BERT) for better understanding of complex queries.
- Support multi-intent queries such as "gaming laptop under 70000 or i5 office laptop under 60000".

2. **Enhanced Product Ranking Algorithm**
   - Introduce **weighted scoring** or **cosine similarity-based matching** to improve accuracy of product recommendations.
   - Allow **user feedback learning** to refine ranking over time.
3. **Database Upgrade**
   - Migrate from SQLite to **MySQL or PostgreSQL** for better scalability and concurrent user handling.
   - Implement migrations using **Flask-Migrate** for structured schema evolution.
4. **Frontend Improvements**
   - Add **filter panels** (price, brand, category) for hybrid search (manual + NLP).
   - Include **pagination**, **sorting**, and **interactive charts** for analytics on user searches.
5. **Deployment & Integration**
   - Deploy on **cloud platforms (AWS, Render, or Heroku)** for public access.
   - Integrate REST API with a **React/Angular frontend** for improved performance and modern UI.
6. **User Personalization and History Tracking**
   - Store user search history and implement **personalized recommendations**.
   - Add a **bookmark/favorite** feature to revisit products later.
7. **Security and Authentication Enhancements**
   - Implement **OAuth or JWT-based authentication** for better session management.
   - Add **password reset via email** and **two-factor authentication** for improved security.
8. **Mobile Application Extension**
   - Develop a **mobile-friendly API** and Android/iOS app version using Flutter or React Native for accessibility.

Final Remarks

The Personalized Product Finder stands as a proof-of-concept that bridges Natural Language Processing and E-Commerce Recommendation Systems in a simple yet effective way.
It can serve as a foundational framework for future research and commercial applications where human-like understanding of user queries can drastically improve user experience and product engagement.

CHAPTER 7 — REFERENCES

7.1 Websites and Online Tutorials

1. Flask Official Documentation — https://flask.palletsprojects.com
2. Flask-Login Documentation — https://flask-login.readthedocs.io
3. Flask-SQLAlchemy Documentation — https://flask-sqlalchemy.palletsprojects.com
4. W3Schools Flask Tutorial — https://www.w3schools.com/flask
5. GeeksforGeeks – Regular Expressions in Python — https://www.geeksforgeeks.org/python-regex
6. Jinja2 Template Documentation — https://jinja.palletsprojects.com
7. Bootstrap 5 Documentation — https://getbootstrap.com/docs/5.3
8. Postman Learning Center — https://learning.postman.com
9. GitHub: SQLAlchemy ORM Examples — https://github.com/sqlalchemy/sqlalchemy

7.2 APIs and Libraries Used

Library / Tool	Version	Purpose
Flask	3.1.2	Core web framework and routing
Flask-Login	0.6.3	Session and authentication handling
Flask-SQLAlchemy	3.1.1	Database ORM integration
SQLAlchemy	2.0.43	Object-relational mapping
Werkzeug	3.1.3	Secure password hashing and verification
Bootstrap	5.3.2	Frontend styling and responsive layout
Jinja2	3.1.6	HTML templating engine

Blinker	1.9.0	Signal management for Flask
itsdangerous	2.2.0	Cryptographic session security
Colorama	0.4.6	CLI color formatting
Typing-Extensions	4.15.0	Python type compatibility
Postman	—	REST API testing and validation
Python	3.11	Programming language used for entire backend

Table 7: Libraries Used

7.3 Development Tools and Platforms

1. **IDE / Code Editor:** Visual Studio Code (v1.94)
2. **Database Tools:** SQLite Studio / DB Browser for SQLite
3. **Testing Tool:** Postman API Client (v11.7)
4. **Diagram Tool:** StarUML (v6.1) for UML design
5. **Documentation & Design:** Microsoft Word 2021, Draw.io for UML exports
6. **Version Control:** Git and GitHub for source management
7. **Operating Environment:** Windows 11 (x64), Python Virtual Environment

7.4 Datasets and Sources

- **Product Dataset:**
  A manually created dataset (products.json) containing entries of electronics and fashion products, used for NLP-based recommendation testing.
  Data fields include category, sub-category, brand, price, features, and use cases.
- No external or commercial datasets were used — ensuring the dataset is entirely original and designed for testing the application's NLP capabilities.